Universidad Nacional Autónoma de México

Facultad de Ciencias

Pantera

Un programa para ilustrar algoritmos de teoría de gráficas

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
ROBERTO ANZALDUA GIL

DIRECTOR DE TESIS:
DRA. ELISA VISO GUROVICH

2010

Hoja de Datos del Jurado

1. Datos del alumno
   Anzaldua
   Gil
   Roberto
   55 30 92 95
   Universidad Nacional Autónoma de México
   Facultad de Ciencias
   Ciencias de la Computación
   302084650

2. Datos del tutor
   Dra
   Elisa
   Viso
   Gurovich

3. Datos del sinodal 1
   Dra
   María de Luz
   Gasca
   Soto

4. Datos del sinodal 2
   Dr
   José de Jesus
   Galaviz
   Casas

5. Datos del sinodal 3
   Dr
   Favio Ezequiel
   Miranda
   Perea

6. Datos del sinodal 4
   M en C
   Liliana
   Reyes
   Cabello

7. Datos del trabajo escrito
   Pantera, un programa para ilustrar algoritmos
   de teoría de gráficas
   81 p.
   2010

Índice general

Prefacio

En este texto presentaremos un paquete llamado llamado Pantera que ilustra la ejecución de algoritmos en gráficas, una introducción a conceptos básicos de teoría de gráficas, un resumen de los algoritmos ilustrados en el sistema, un manual de cómo usar el programa, un manual para el programador interesado en agregar algoritmos y algunos detalles de implementación.

El paquete está desarrollado usando el lenguaje de programación java y el programa para visualización de gráficas graphviz e ilustra los siguientes algoritmos:

- Algoritmo BFS
- Algoritmo de Circuitos eulerianos
- Algoritmo DFS
- Algoritmo de distancias de Dijkstra

Los dos objetivos principales que cumple este trabajo son:

- Apoyar el aprendizaje de algoritmos de teoría de gráficas mediante un programa debidamente documentado, en donde se muestra la ejecución detallada de algoritmos de teoría de gráficas sobre una gráfica diseñada por el usuario. En la ejecución se muestra: el paso actual, una descripción de lo ocurrido durante cada paso y una imagen de la gráfica en la que se resaltan los cambios durante el desarrollo del algoritmo
- Permitir programar algoritmos ilustrados de teoría de gráficas que funcionen como lo hacen los ya incluidos en el paquete a cualquier persona que programe en java.

La motivación para el desarrollo de este sistema comenzó con la idea de ayudar a los alumnos del curso de matemáticas discretas a estudiar los algoritmos de teoría de gráficas, pero conforme se fue avanzando con el sistema, nos dimos cuenta de que serviría para cualquier curso en donde se revisen algoritmos de teoría de gráficas y que además es posible agregar algoritmos sin mucha dificultad a la adicional de programarlo. Por otra parte, en los últimos años, las tecnologías asociadas a la programación han permitido crear herramientas didácticas para la enseñanza de todas las disciplinas, facilitando el aprendizaje de temas complicados y motivando al alumno a conocer más; sin embargo, a pesar de que teoría de gráficas es un tema que se usa en prácticamente todas las ramas de ciencias de la computación y que se estudia en diversas materias de carreras como matemáticas, actuaría, física, y algunas ingenierías, no hay herramientas en español sencillas de usar que no involucren más conocimiento que el básico en el tema, y la mayoría de las que existen se limitan a ilustrar sólo un algoritmo y mediante ejemplos particulares. En el caso de este programa, es posible diseñar la gráfica con la que se va a trabajar e incluso programar un nuevo algoritmo.

Parte I

Antecedentes y fundamentos

Capítulo 1

Introducción y conceptos básicos

El estudio de teoría de gráficas es una rama de las matemáticas que tiene sus raíces hace poco más de dos siglos cuando Leonard Eüler en 1736 comenzó con las primeras ideas. En su trabajo resolvía un problema muy famoso de su época que se conoce como *el problema de los puentes de Königsberg* que es considerado el primer resultado del tema y del cual hablaremos más adelante.

En 1852, Francis Guthrie formuló el problema de los cuatro colores que planteaba si era posible colorear cualquier mapa de países usando sólo cuatro colores y de tal forma que países vecinos nunca tengan el mismo color. Este problema fue resuelto hasta 1976 por Kenneth Appel y Wolfgang Haken quienes crearon un programa que tardaron cuatro años en terminar. El sistema fue ejecutado en la computadora Cray y a ésta le tomó 1200 horas revisar 1476 configuraciones.

Esta demostración generó una controversia para el mundo de las matemáticas pues la prueba se realizó usando una computadora. Los matemáticos se sintieron incrédulos ante el resultado, pues dado que se puede explicar muy fácilmente el teorema, ellos piensan que la demostración debería ser posible de llevarse a cabo por un humano de alguna manera satisfactoria.

Además de probar el resultado de teoría de gráficas, se constató la utilidad y la importancia actual de las computadoras para el trabajo científico, ya que durante más de un siglo se intentó probar el teorema sin éxito.

A todos los avances tecnológicos que se han tenido se les debe un resurgimiento en el interés en la teoría de gráficas, con la cual no sólo es posible resolver problemas relacionados con ciencias de la computación, sino también es posible resolver problemas de negocios y relacionados con otras ciencias; es por esto que se ha convertido en un tema importante en muchos ámbitos del conocimiento.

1.1. Conceptos y representación de gráficas

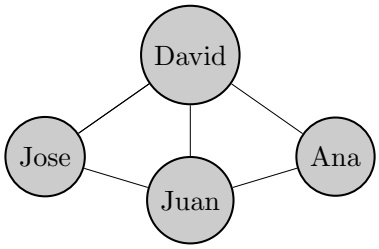
Muchas situaciones de la vida real pueden ser modeladas mediante un diagrama que consiste en un conjunto de puntos y segmentos que unen ciertos pares de puntos entre los cuales existe una relación. Por ejemplo:

- Los puntos pueden representar personas y los segmentos unirían amigos (como lo podemos ver en la figura 1.1).

- Los cruces de calles pueden representarse con puntos y se pondría un segmento que una dos cruces por cada camino que exista entre ellos.

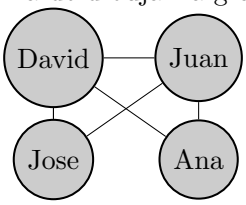
- Los puntos pueden representar ciudades y habría un segmento entre cada par de ciudades que tengan un vuelo sin escalas entre ellas.

Figura 1.1: Personas relacionadas con sus amigos



Tal vez algunas de estas situaciones les parezcan familiares, pues es una idea muy natural e intuitiva que se usa para representar una infinidad de problemas. En todos estos casos hay que observar que lo que nos interesa de dichos diagramas es el hecho de si existe una línea uniendo cierto par de puntos. La idea general de los ejemplos se puede representar mediante un dibujo de un conjunto de objetos en el cual algunos pares de estos objetos están relacionados; esta representación es sencilla de comprender y trazar sobre el papel; sin embargo, es necesario tener una representación escrita en el lenguaje matemático para poder referirnos de manera única a gráficas iguales. Existe más de una forma de trazar la misma gráfica, por ejemplo, la gráfica en la figura 1.1 se puede dibujar como en la figura 1.2:

Figura 1.2: Otra forma de dibujar la gráfica de la figura 1.1

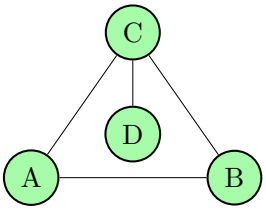


Por otra parte, es difícil "pintar" una gráfica y usar la computadora para que trabaje con el dibujo, por lo que es necesario definir formalmente a este tipo de objetos.

Una **gráfica** G es un conjunto de vértices o nodos, que denotamos por $V(G)$ (los 'puntos' como lo describimos antes), y un conjunto de parejas de dos elementos de $V(G)$ que se conocen como los arcos o aristas de la gráfica (los 'segmentos' en los dibujos), y que usaremos $E(G)$ para representarlo[1]. Por otra parte, una **subgráfica** $G' = (V', E')$ de G es una gráfica tal que $V' \subseteq V$ y $E' \subseteq E$. Si u y v son vértices de la gráfica y están relacionados, habrá una arista que los une y usaremos uv para denotarla. Hay que observar que para fines de este trabajo las aristas serán parejas *no ordenadas*, es decir uv y vu representarán a la misma arista. En la figura 1.3 se muestra un ejemplo para hacer más clara la definición:

[1] Cuándo estemos trabajando sólo con una gráfica, podemos escribir simplemente V para denotar a su conjunto de vértices o E para el conjunto de aristas

Figura 1.3: Dibujo de una gráfica y los conjuntos de vértices y aristas que la definen



$G = (V, E)$

$V(G) = \{A, B, C, D\}$

$E(G) = \{AB, AC, BC, CD\}$

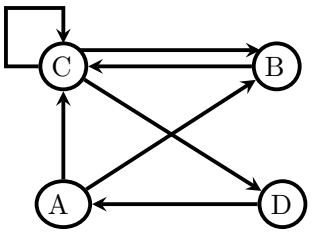
Si uv es una arista, tendremos que la arista *incide* en u y en v, que u es adyacente a v y que v es adyacente a u, también diremos que u es vecino de v y que v es vecino de u.

Algunas de las características que una gráfica puede presentar son las siguientes:

1. **Lazos.** Es una arista cuyos extremos son el mismo vértice (vv).
2. **Aristas múltiples.** Decimos que una gráfica tiene aristas múltiples cuando una pareja de vértices tiene más de una arista que los relaciona; a las gráficas que poseen este tipo de aristas se les conoce como multigráficas.
3. **Gráficas infinitas.** La gráfica presenta un número infinito de vértices o aristas.
4. **Aristas dirigidas.** Son las aristas que tienen una *dirección*, es decir, si $e = uv$ y $e_1 = vu$, $e \neq e_1$. A las gráficas con *aristas dirigidas* se les conoce como *digráficas*.

A continuación, en la figura 1.4 un ejemplo de algunas de las características mencionadas

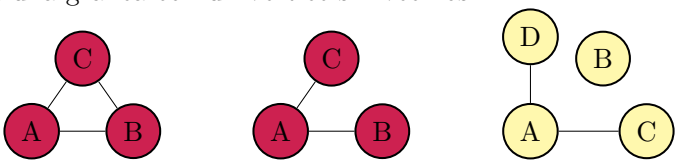
Figura 1.4: Una digráfica con un lazo en el vértice C.



Las gráficas con dichas características, no se encuentran dentro del alcance de este texto. Nosotros estaremos trabajando con gráficas sin lazos, sin aristas múltiples, no dirigidas, con un número finito de vértices -pero que al menos tenga un vértice-, pues los algoritmos ilustrados por el paquete así lo requieren. A estas gráficas se les conoce como *gráficas simples no dirigidas*; son las que usaremos a menos que especifiquemos lo contrario, por lo que las llamaremos simplemente gráficas.

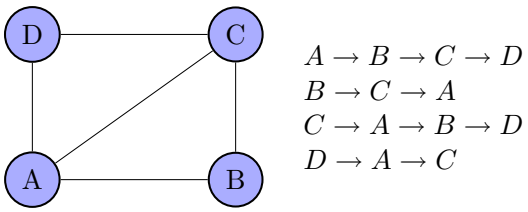
Hay que observar que ambos conjuntos, el de vértices y el de aristas, son necesarios para definir una gráfica; es imposible deducir correctamente uno mediante el otro. En el caso de los vértices se entiende que no podemos saber cuáles son las aristas pues existen muchos posibles conjuntos de aristas para un mismo conjunto de vértices. Si sólo conociéramos las aristas, podríamos deducir a todos aquellos vértices que tienen vecinos, pero es posible que existan vértices sin vecinos. En la figura 1.5 podemos ver un ejemplo de lo que acabamos de mencionar.

Figura 1.5: A la izquierda se muestran dos gráficas con el mismo número de vértices y un conjunto distinto de aristas y a la derecha una gráfica con un vértice sin vecinos



Existen varias formas de representar una gráfica G, la que nos interesa es la conocida como representación por listas de adyacencias, en donde a cada vértice le corresponde una lista que contendrá a todos aquellos vértices con los que está relacionado, es decir tendremos $n = |V(G)|$ listas. Estas listas pueden estar contenidas en un vector en el cual la lista correspondiente al i-ésimo vértice sea el elemento en la posición i del vector. Cada elemento de la lista tiene una referencia[2] al siguiente vértice adyacente y en caso de ser el último, tiene una referencia nula que quiere decir que ya no hay mas elementos adyacentes. A continuación, en la figura 1.6, se muestra un ejemplo de una gráfica representada con sus listas de adyacencias:

Figura 1.6: Representación de una gráfica con listas de adyacencias



Lo que está haciendo esta representación es tomar cada vértice y "anotar" cuáles son sus vecinos. Esta representación nos resulta práctica pues en cada lista de adyacencia, existe exactamente un elemento por cada vértice adyacente.

El número de aristas que inciden en un vértice v se define como el *grado* de v y usaremos $grado(v)$ para referirnos a él. Por ejemplo, en la figura 1.6 el vértice D y el B, tienen grado 2 mientras que $grado(C) = 3$ y $grado(A) = 3$.

La definición de gráfica es sencilla, por lo que ha sido muy conveniente que las gráficas puedan ser utilizadas para modelar diversos problemas o ayudar a resolverlos, por ejemplo:

- Decidir si es posible recorrer varios lugares sin pasar por el mismo punto dos veces.
- Encontrar el camino más corto entre dos puntos de una ciudad.
- Encontrar la cantidad mínima de cableado necesario para un edificio de tal manera que todos los pisos estén conectados.
- Asignar tareas de distinto costo a personas con distintas habilidades de la manera más eficiente posible.
- Determinar el flujo que circula por una tubería.

Los problemas anteriores son resueltos de forma habitual alrededor del mundo y es por ello que la teoría de gráficas es muy importante. Todos requieren realizar una exploración en una gráfica, es decir

[2] En términos de programación, se le conoce también como apuntador o liga

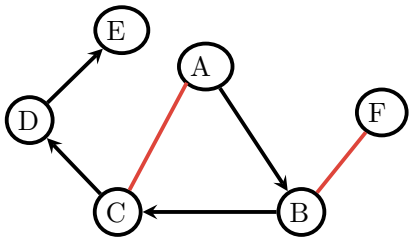
recorrerla para obtener información relevante de acuerdo a nuestras necesidades, por lo que ahora nos concentraremos en revisar problemas de exploración.

1.2. Exploración de gráficas

Para realizar una exploración en una gráfica, es necesario plantearse dónde vamos a comenzar, la manera en la que decidiremos hacia dónde seguir y cuándo vamos a parar. Conforme nos movemos sobre la gráfica, iremos generando un *camino*, que probablemente almacenaremos para decidir si es momento de detenerse. Un **camino** en una gráfica es una sucesión de vértices intercalados con aristas: $v_1, e_1, v_2, e_2, \ldots, e_n, v_{n+1}$, donde si $u = v_1$ y $v = v_{n+1}$ entonces tenemos un camino entre u y v también llamado **camino-**uv, cada arista e_i incide en v_i e incide en v_{i+1}, diremos también que v_i es el predecesor de v_{i+1} para todo i en $1, \ldots, n$. Por otro lado, la **longitud** de un camino C se define como el número de aristas contenidas en el camino y se usa $|C|$ para denotarlo. Usaremos $u \rightsquigarrow v$ cuando nos estemos refiriendo a un camino del vértice u al vértice v que no necesariamente tiene longitud uno y cuando necesitemos especificar que el camino sólo contiene una arista, usaremos: $u \to v$. Cómo estamos trabajando con gráficas sin aristas múltiples, cuando definamos un camino, no siempre será necesario escribir sus aristas, es decir, si $C = v_1, e_1, v_2$ es un camino, podemos escribirlo también como $C = v_1, v_2$.

En la figura 1.7 se ilustra un camino del vértice A al vértice E marcado con flechas negras, este camino tiene cuatro aristas, por lo que su longitud es cuatro.

Figura 1.7: Una gráfica con un camino

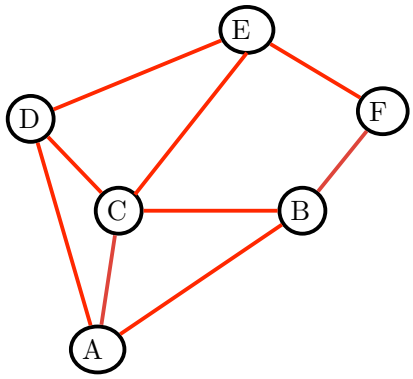


Usamos flechas sólo para remarcar que el camino fue escogido sin pérdida de generalidad de A a E, pero no estamos tratando con aristas dirigidas, por lo que, del camino-AE se puede obtener un camino-EA comenzando en E y siguiendo todas las aristas que usamos para el camino-AE.

Ahora que conocemos la definición de camino, podemos intentar resolver problemas más complicados, por ejemplo, supongamos que tenemos ciertas computadoras que están conectadas mediante una serie de cables y que deseamos eliminar algunos de ellos ya que son estorbosos y quizá innecesarios, pero conservando la comunicación entre todas las computadoras. Para modelar el problema usando teoría de gráficas, cada computadora sería un vértice y habría una arista entre cada par de computadoras conectadas mediante un cable.

En la figura 1.8 se muestra una gráfica que representa una red de computadoras:

Figura 1.8: Una gráfica que representa una red de computadoras



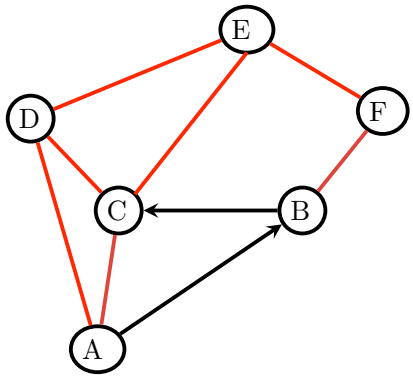
Si una computadora A tiene un cable que la conecta con una computadora B, y ésta tiene un cable que la conecta con una computadora C, A se puede comunicar con C pues A puede acceder a toda la información que puede ver B y eso incluye a la información de la computadora C.

De lo anterior, podemos notar que en nuestro modelo, una computadora se puede *comunicar* con otra si existe un *camino* entre ellas; nos interesaría que entre cualesquiera dos computadoras tengamos un camino para que se pudieran comunicar. Es exactamente esto lo que queremos preservar, la capacidad de comunicación entre las computadoras.

Una estrategia sería quitar algunos cables y probar que la condición de conexión se cumpla para todas las computadoras. El problema de esta estrategia es que quitar cables al azar no es eficiente y verificar que se preserve la conexión entre todas las computadoras es bastante costoso; lo que necesitamos hacer es encontrar una manera en la que seleccionemos cables bajo algún criterio con el que estemos seguros de que las computadoras siguen preservando la propiedad antes mencionada.

En la figura 1.9 se muestra una gráfica con un camino demostrando que la computadora A se puede comunicar con la computadora C:

Figura 1.9: Una gráfica que representa una red de computadoras

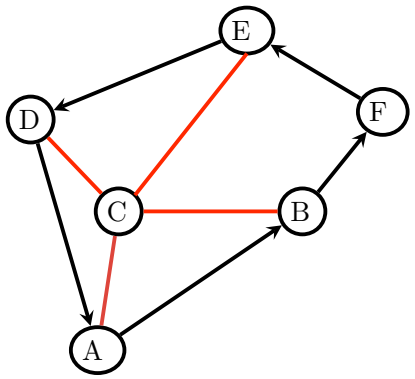


Continuando con la misma gráfica, podemos observar un camino de A a E, conteniendo $P_1 = A, B, F, E$; y otro de E a A; $P_2 = E, D, A$. De estos dos caminos podemos obtener un ciclo: $P' = A, B, F, E, D, A$.

Formalmente, un **ciclo C** es un camino **cerrado**, es decir un camino que comienza y termina en el mismo vértice. Un **ciclo simple** es un ciclo que no repite aristas y en el que sólo el primer y último vértice son el mismo. Dado que usar la definición general de ciclo nos podría causar problemas en algunas demostraciones, cuando hablemos de un ciclo, nos estaremos refiriendo a un ciclo simple, a menos que especifiquemos lo contrario. Cuando tenemos un ciclo en una gráfica, entre cual quier par de vértices contenidos en el ciclo, existen al menos dos caminos: el que toma la dirección del ciclo y el que toma la dirección contraria. Por ejemplo, para los vértices A y E, podemos considerar el camino P_1 e invertir el camino P_2 para obtener el camino $P_3 = A, D, E$.

El ciclo formado usando los elementos de P_1 y P_2 se muestra en la figura 1.10.

Figura 1.10: Ejemplo de ciclos en una gráfica



Una gráfica es **conexa** si para todo par de vértices $u, v \in V(G)$ existe un camino de u a v. Esta es una de las definiciones más importantes en los conceptos de exploración de gráficas, pues lo que queremos, en general, es poder acceder por todos los vértices al menos una vez para obtener datos importantes; si la definición anterior no se cumple, habrá vértices inalcanzables y diremos que la gráfica es **disconexa**. Un vértice $v \in V(G)$ es **inalcanzable** desde $s \in V(G)$ si no existe existe un camino entre ellos, por el contrario, si existe un camino entre ellos, definimos v como **alcanzable** desde s.

Podemos observar que si la gráfica en el problema de las computadoras en red es conexa, entonces garantizamos que todas las computadoras se pueden comunicar entre ellas; lo que buscaremos es quitar tantas aristas de la gráfica como sea posible, pero asegurando que la gráfica siga siendo conexa. Introducimos la siguiente operación: $G - e$ representa a la subgráfica obtenida al eliminar la arista e de G.
Tenemos la siguiente afirmación:

Afirmación 1 *Si G es una gráfica conexa y $e \in G$ es una arista contenida en un ciclo, $G - e$ es una gráfica conexa.*

Demostración:

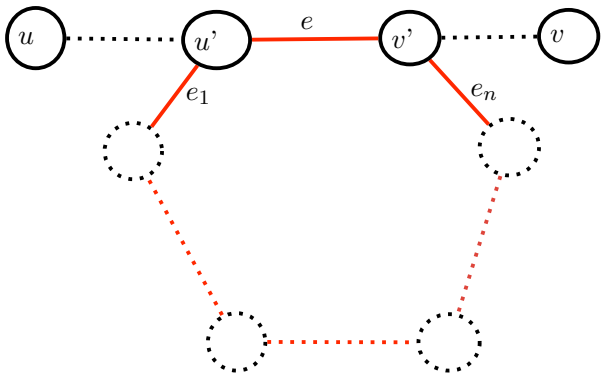
Necesitamos ver que para cualesquiera $u, v \in V(G)$ es posible encontrar un camino C' en $G - e$ que va de u a v. Sea C el camino-uv existente en G. La demostración construirá a C' en $G - e$ a partir de C, así que podemos dividir en dos casos la demostración:

Caso 1: la arista e no está contenida en C. En este caso $C \in G - e$ por lo que $C' = C$.

Caso 2: la arista e está en C. Ahora es necesario *redirigir* el camino a partir del vértice en uno de los extremos de e como lo hicimos antes. Sea $C = u, \ldots, u', e, v', \ldots, v$ el camino-uv que existe en G, como e está contenida en un ciclo, u', v' también; como vimos anteriormente existen al menos dos caminos entre u' y v' que están contenidos en el ciclo, uno de ellos es $C_1 = u', e, v'$; sea $C_2 = u', e_1, \ldots, e_n, v'$ el otro camino que no contiene a e. Si sustituimos u', e, v' por C_2 en C obtendremos $C' = u, \ldots, u', e_1, \ldots, e_n, v', \ldots, v$ que es el camino que estamos buscando.

La figura 1.11 ilustra como tomar un camino que vaya de u' a v' y que no pase por e.

Figura 1.11: Redirigir el camino



■

Con lo anterior ya tenemos un buen criterio para quitar aristas de una gráfica, si quitamos una arista contenida en un ciclo, nuestra gráfica permanecerá conexa. Dicho de otra manera, siempre que encontremos un ciclo en una gráfica conexa, es posible quitar al menos una arista y la gráfica seguirá siendo conexa.

En la afirmación 1 es posible que el camino generado contenga vértices y aristas repetidas, pues en el caso especial en el que u, v estén contenidos en el ciclo en el que está e, repetiremos un pedazo del ciclo y nos regresaremos por donde ya pasamos. En general, nos gustaría no repetir vértices pues no hay razón para pasar por el mismo lugar dos veces, si resulta que u, v están en el mismo ciclo que e, sería conveniente tomar el camino que va en la otra dirección del ciclo y que no pasa por e.

A los caminos que no repiten vértices y por lo tanto no repiten aristas se les conoce como **trayectorias**.

La definición de trayectoria puede hacer más conveniente la definición de gráfica conexa, para ello demostramos lo siguiente:

Afirmación 2 *Todo camino-uv contiene una uv-trayectoria.*

Demostración:

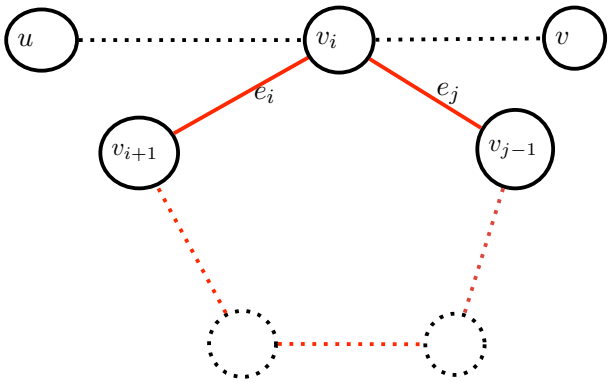
Sea C un camino-uv. Si C no contiene vértices repetidos, por definición se trata de una trayectoria.

En otro caso, C contiene vértices repetidos y por lo tanto, podemos escribirlo de la siguiente manera: $C = v_1, e_1, \ldots, v_i, e_i, v_{i+1}, e_{i+1}, \ldots, e_{j-1}, v_j, e_j, \ldots, e_{n-1}, v_n$ en dónde $v_i = v_j$.

Si eliminamos $e_i, v_{i+1}, e_{i+1}, \ldots, e_{j-1}, v_j$ de C, nos quedaremos con $v_1, e_1, \ldots, v_i, e_j, \ldots, e_{n-1}, v_n$ que también es un camino, si dicho camino ya no contiene vértices repetidos, ya encontramos la trayectoria contenida en C; de no ser así, volvemos a repetir el proceso hasta que obtengamos la trayectoria. Dado que el número de vértices es finito, podemos asegurar que finalmente obtendremos una trayectoria.

La figura 1.12 ilustra cómo se ve un camino con vértices repetidos.

Figura 1.12: Eliminar ciclos



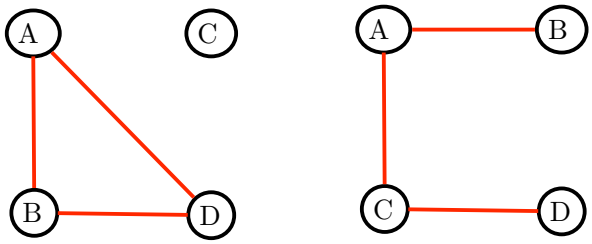
■

Para cada par de vértices u, v contenidos en una gráfica conexa, existe una trayectoria-uv. La siguiente pregunta que nos podemos hacer es: ¿cuándo una gráfica contiene ciclos? La respuesta por supuesto tiene que ver con el número de aristas de la gráfica y las relaciones entre los distintos vértices.

Por ejemplo, en la figura 1.13 se muestran dos gráficas con el mismo número de aristas y el mismo número de vértices, pero la gráfica de la derecha es conexa, y la gráfica de la izquierda está dividida en dos componentes conexas: el componente que contiene a los vértices A, B, D y el que contiene a C.

Sea G una gráfica. Una subgráfica H de G es una **componente conexa** de G si H es conexa y además H es *máxima por contención* respecto a la propiedad de ser conexa. Dicho de otra manera, una subgráfica conexa H de G es una componente conexa de G si $H = H'$ para toda H' subgráfica de G que contiene a H.

Figura 1.13: Gráficas con el mismo número de aristas y el mismo número de vértices



La siguiente afirmación nos ayudará a comprender mejor la relación que existe entre el grado de los vértices y los ciclos que contiene:

Afirmación 3 *Sea G una gráfica. Si para cada vértice $v \in V$ se cumple que $grado(v) \geq 2$, entonces G contiene al menos un ciclo.*

Demostración:

En efecto, si tomamos un vértice s de G y comenzamos un camino, dado que existen al menos dos aristas que inciden en cada vértice, siempre podremos salir de él y continuar el camino. Puesto que el número de vértices en la gráfica es finito, eventualmente regresaremos al vértice inicial generando un ciclo.

■

Ahora hemos llegado a una pregunta que es necesario responder para poder conocer la respuesta de la anterior: ¿cuántas aristas tiene una gráfica conexa que no tiene ciclos?

Una idea es realizar un proceso de quitar aristas de los ciclos de la gráfica hasta que el número de aristas sea menor que el de vértices. Por ejemplo, la componente conexa de la gráfica que aparece del lado izquierdo de la figura 1.13 forma un ciclo que contiene el mismo número de vértices que de aristas y claramente son más de las necesarias, pues si quitamos cualquier arista la gráfica sigue siendo conexa, pero al quitar más de una arista, la gráfica se desconecta.

Lo anterior nos sugiere intentar una demostración aseverando que a partir de una gráfica conexa G, es posible crear una G' quitando aristas contenidas en los ciclos de G hasta que ya no queden más ciclos y entonces tendríamos que $|E(G')| = |V(G')| - 1$ y como demostramos en la afirmación 1, G' seguirá siendo conexa.

Vamos a demostrar que una gráfica conexa y sin ciclos tiene al menos dos vértices de grado uno, y que si le quitamos alguno de dichos vértices la gráfica resultante sigue siendo conexa. Antes de formalizar este resultado, introducimos las siguientes operaciones para una gráfica G, un camino $C = v_1, \ldots, v_n$ y un vértice v:

- Al escribir $G - v$, denotamos a la gráfica resultante de quitar el vértice v de G y a todas las aristas que inciden en el vértice.
- Cuando decimos $C \cup v$ nos referimos a crear un nuevo camino $C' = v_1, \ldots, v_n, v$

Afirmación 4 *Si G es una gráfica conexa y sin ciclos, entonces G contiene al menos dos vértices de grado uno.*

Demostración:

Sea $T = v_1, v_2, \ldots, v_{n-1}, v_n$ una trayectoria de longitud mayor contenida en G. Demostraremos que v_1 y v_n tienen grado uno. Lo haremos por contradicción comenzando con v_1.

Claramente, el grado de cada vértice en G es al menos 1, pues la gráfica es conexa. Como estamos suponiendo que $grado(v_1) \neq 1$, tenemos que $grado(v_1) > 1$, de aquí que es adyacente al menos a un vértice v distinto de v_2. Tenemos dos casos posibles para este vértice:

- $v \in T$. Si esto sucede, entonces podemos usar la trayectoria contenida en T que va de v_1 hasta v y luego regresar a v_1, generando el ciclo $C = v_1, v_2, \ldots, v, v_1$ que no es posible porque por hipótesis G no tiene ciclos.

- $v \notin T$. Aquí es posible construir la trayectoria $T' = T \cup v$, que es de longitud mayor a T, lo cual contradice nuestra primera suposición.

Acabamos de demostrar que el $grado(v_1)$ no es mayor que 1, entonces $grado(v_1) = 1$. Demostrar que v_n es un vértice de grado uno es análogo a lo anterior, por lo que concluimos que v_1 y v_n tienen grado uno.

■

Afirmación 5 *Sea G una gráfica conexa con n vértices y sin ciclos. Entonces G tiene $n-1$ aristas.*

Demostración:

Por inducción sobre el número de vértices.

Observación. Podemos utilizar de caso base cuando el número de vértices es 1, pero ese caso es trivial pues dicha gráfica tiene cero aristas y es conexa, por lo que empezaremos en el caso en el que $|V(G)| = 2$.

Caso base. Empezaremos con $|V(G)| = 2$. Como estamos trabajando con gráficas simples, sin lazos, y por hipótesis conexa, estos vértices tienen una arista que los une, por lo que $|E(G)| = 1$.

Figura 1.14: La única gráfica conexa posible de dos vértices

Hipótesis de inducción. Si G contiene $n-1$ vértices, es conexa y sin ciclos, entonces G contiene $n-2$ aristas.

Paso inductivo. Sea G una gráfica con n vértices, conexa y sin ciclos. Tenemos que demostrar que G contiene $n-1$ aristas.

Por la afirmación 4, G contiene dos vértices de grado uno; sea v uno de esos vértices. Consideremos a $G' = G - v$; como sólo quitamos un vértice y una arista, G' tiene $n-1$ vértices y esta gráfica cumple con la hipótesis de inducción por lo que G' tiene $n-2$ aristas. El número de aristas de G es igual a $|E(G')| + 1 = n - 2 + 1 = n - 1$.

■

Un **árbol** es una gráfica conexa y sin ciclos. Un **árbol generador** de una gráfica G es una subgráfica conexa de G que contiene todos sus vértices y que tiene exactamente $n-1$ aristas. Como veremos a continuación, todas las gráficas poseen un árbol generador:

Corolario 1 *Toda gráfica conexa G tiene un árbol generador G'.*

Demostración:

De las afirmaciones 4 y 5 se deduce que para cualquier gráfica G conexa con n vértices y k aristas, es posible obtener otra G' que contenga $n-1$ aristas y que sea conexa, basta con quitar una arista de algún ciclo y repetir el proceso hasta que G ya no tenga ciclos. Como G' así obtenida es conexa y acíclica, es un árbol; y como no quitamos ningún vértice, es un árbol generador.

■

Los árboles generadores son muy importantes en teoría de gráficas, porque si se obtuvieron de gráficas complejas, se tiene una estructura en donde están todos los vértices y en la cual es posible encontrar una trayectoria entre cualquier par de vértices. Los árboles son estructuras frágiles, como veremos a continuación: eliminar una arista de un árbol provoca que la gráfica se desconecte:

Afirmación 6 *Si G un árbol y $e \in E(G)$; entonces $G' = G - e$ no es conexa.*

Demostración:

Haremos la demostración por contradicción.

Sea $G' = G - e$ y $e = v_i v_j$. Supongamos que G' es conexa; entonces existe una trayectoria $T = v_i, \ldots, v_j$ en G', como G' está contenida en G, $T \in G$, por lo que podemos considerar a $C' = T \cup v_i = v_i, \ldots, v_j, v_i$ que es un ciclo en G. Esto no es posible porque G es un árbol.

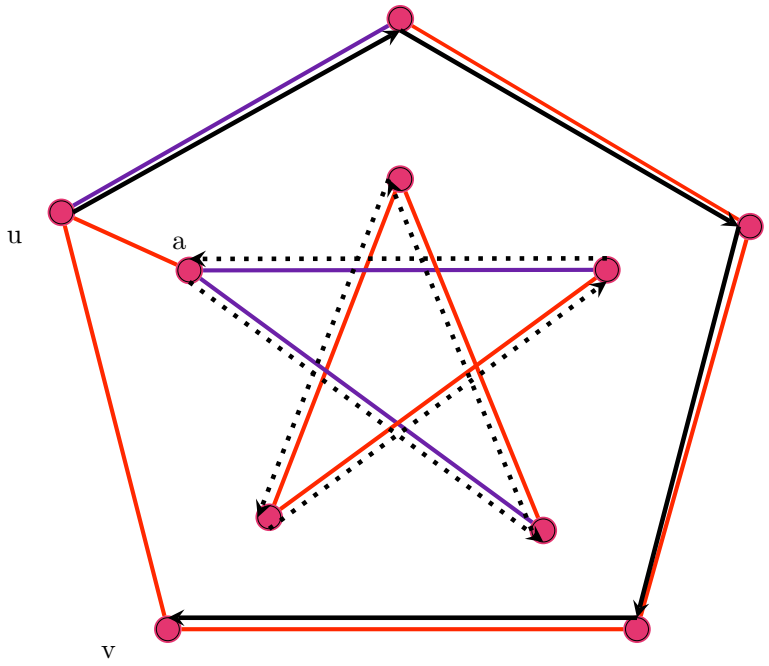
■

Otro problema común radica en encontrar alguna forma de desplazarse desde un punto de la ciudad hacia otro. Como hemos mencionado, esto lo podemos modelar utilizando teoría de gráficas. Los vértices serían los cruces de las calles y entre cruces se dibujaría una arista por cada calle que podemos recorrer. Intuitivamente, para encontrar un camino iríamos marcando las aristas por las que pasamos hasta el momento en el que lleguemos al vértice deseado.

El problema de hacerlo sin restricción alguna y escogiendo la siguiente arista arbitrariamente, es que generaríamos una ruta que difícilmente sería la mejor; por ejemplo, cuando vamos recorriendo la ciudad en carro, nos ha pasado que por desconocer el lugar a donde vamos pasamos por el mismo lugar más de una vez; en tal caso, nuestro camino tendría vértices repetidos generando un ciclo. Aunque lleguemos al mismo lugar, la ruta que generamos no es la más corta. Si tenemos un camino $C = v_1, \ldots, v_i, v_{i+1}, \ldots, v_j, v_{j+1}, \ldots, v_n$ en donde $v_i = v_j$, dicho camino contiene al ciclo $C' = v_i, v_{i+1}, \ldots, v_j$, que puede ser eliminado de C y llegaríamos a donde queremos sin tener que pasar por C'. Si después de quitar C' de C continúan existiendo vértices repetidos, entonces podemos volver a repetir el proceso hasta que nuestra ruta no tenga ciclos, generando una trayectoria, que es en general lo que buscamos tener para no pasar por el mismo lugar dos veces.

Por ejemplo, en la figura 1.15 se muestra un camino que también es trayectoria entre u y v, denotado por líneas sólidas, y un ciclo que comienza en el vértice a, que está denotado por líneas punteadas; si quisiéramos ir de a a v, podríamos alargar el camino y seguir por el ciclo que comienza en a, o simplemente ir de a a u y recorrer la trayectoria de u a v.

Figura 1.15: Ciclos y trayectorias



Después de introducir los conceptos básicos de teoría de gráficas, revisaremos algunas convenciones y definiciones relacionadas con algoritmos.

1.3. Representación de algoritmos

En las siguientes secciones se usará **pseudo-código** para describir los algoritmos que se revisarán. El **pseudo-código** es una descripción de un algoritmo que emplea la mezcla del lenguaje natural con algunas convenciones de programación.

Los algoritmos usarán alguna estructura para almacenar los resultados, como por ejemplo una **cola**, que es una estructura de datos parecida a una lista en la que el primer elemento que puede ser obtenido, es aquel que se agregó primero y el último que se obtendrá será el que se agregó al final. Para una cola $C = [v_1, \ldots, v_n]$ tenemos las siguientes operaciones:

- **Agregar.** Esta operación permite agregar elementos a la cola. Siempre se agregan al final, por ejemplo, agregar el elemento v la cola provocaría que $C = [v_1, v_2, \ldots, v_n, v]$.
- **Pedir un elemento.** Cuando pedimos un elemento a la cola, ésta nos regresará el elemento que tenga al frente, en este caso nos regresaría a v_1.
- **Eliminar un elemento de la cola.** Aquí, al ejecutar esta operación nos dejará con $C = [v_2, \ldots, v_n, v]$.

También usaremos una **cola de prioridades** para el algoritmo de Dijkstra. Estas estructuras definen una *prioridad* para cada uno de sus elementos; por ejemplo, al sacar un elemento obtendremos a aquel con mayor **prioridad** dentro de la cola, que no necesariamente será el que esté al frente.

Una **variable** en programación es un elemento del algoritmo que puede cambiar su valor durante la ejecución.

Además de las estructuras de datos, usaremos las siguientes definiciones:

- **Asignación**. Cuando tengamos $v \leftarrow u$ quiere decir que a la variable v se le asigna el valor de u.
- **Atributo**. Si tenemos un elemento v que posee algún **atributo** d, cuando nos refiramos al atributo, usaremos la notación $v.d$. Por ejemplo, la distancia de un vértice la podemos representar con d y usaremos $v.d$ para referirnos a ella. También podemos almacenar estructuras como propiedades; por ejemplo, si tenemos un vértice u y escribimos $u.adyacencia$, nos referimos a la lista de vértices adyacentes a u.
- **Uso de corchetes ([ ])**. Cuando escribamos algo entre corchetes, será para facilitar la lectura de el contenido de una estructura de datos, por ejemplo, si tenemos una cola $C = [v_1, \ldots, v_n]$ queremos decir que la cola C contiene a los elementos $v_1, \ldots, v_n$ en ese órden.
- **Condición.** Es una expresión que se puede evaluar como falsa o verdadera.
- **Ciclo.** Un ciclo es un procedimiento que se repite hasta que se cumple cierta condición.
- **Mientras**. Cuando veamos esta palabra dentro de algún algoritmo, representará el inicio de un ciclo y siempre tendrá a su derecha alguna condición para salir del ciclo y continuar con el resto del algoritmo.
- ==. Es un operador lógico que se aplica a dos elementos. Si tenemos $v == u$ el operador regresará verdadero en caso de que v y u sean iguales y falso en caso contrario; como podemos ver, este operador es una condición.
- **Si**. Cuando veamos esta palabra siempre tendrá a su lado derecho una condición; si la condición se cumple, se ejecutará lo que sigue en las líneas que tiene después.
- **Invariante.** Una invariante es una afirmación que se mantiene durante toda la ejecución de un algoritmo.

Ahora comenzaremos con el desarrollo de los algoritmos.

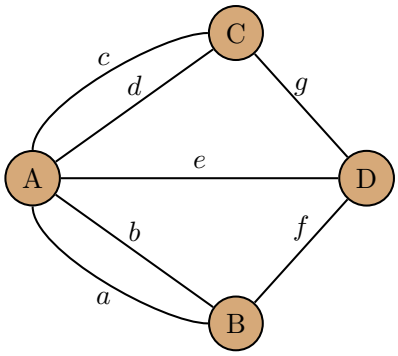
1.4. Circuitos Eulerianos

Hay muchas aplicaciones a problemas de la vida real relacionados con caminos, el primero fue un problema resuelto en el siglo XVIII por Leonard Eüler y surge como solución a un problema sencillo conocido hoy como *circuito euleriano*.

El problema se conoce como los puentes de Königsberg y consiste en lo siguiente:

Dos islas en el río Pregel que cruza Königsberg se unían entre ellas y con la tierra firme mediante siete puentes. El problema radicaba en encontrar un paseo que recorriera cada puente sin volver a pasar por el mismo y terminar donde se empezó. La figura 1.16 es una representación del problema; los vértices simbolizan pedazos de tierra y las aristas a los puentes de la ciudad.

Figura 1.16: Problema de los puentes de Königsberg

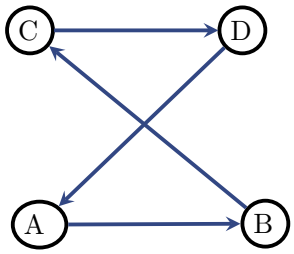


Observando la gráfica, nos podemos percatar que lo que se quiere encontrar es un *ciclo*[3] que pasa por todas las aristas de la gráfica exactamente una vez y termina en el vértice en donde empezó; a estos ciclos se les conoce como **circuitos eulerianos** y de esta definición surge el concepto de **gráfica euleriana**, que es aquella que cumple con tener un circuito euleriano y que dicho circuito contenga todos sus vértices y aristas.

Eüler demostró que una gráfica contiene un circuito euleriano si y sólo si los grados de todos los vértices son pares y podemos observar en la figura 1.16 que la gráfica que representa el problema contiene vértices con grado impar y por lo tanto no contiene un circuito euleriano.

Dado que los circuitos eulerianos trazan una ruta que demuestra que es posible recorrer toda la gráfica, decimos que le estamos asignando dirección a las aristas de la gráfica, en la figura 1.17 se muestra una gráfica con un circuito euleriano:

Figura 1.17: Una gráfica con un circuito euleriano marcado



Decidir si una gráfica es euleriana es importante para muchos problemas, por ejemplo:

- Obtener una ruta para que un recolector de basura no pase por la misma calle dos veces dentro de una colonia.
- Cerrar un circuito en una placa electrónica.
- Seleccionar una ruta para los aviones de una aerolínea.

[3] En el caso de circuitos eulerianos, no necesariamente se tratará de un ciclo simple

Si una gráfica G contiene un circuito euleriano que empieza en un vértice particular $v \in G$, también es posible dar uno para cualquier vértice de G usando el circuito euleriano que ya tenemos. En general, si queremos encontrar un circuito euleriano que comience en v_i y tenemos un circuito euleriano $P = v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n$ de G en el que $v = v_n = v_1$, el ciclo que buscamos es
$P' = v_i, v_{i+1}, \ldots, v_n = v, \ldots, v_{i-1}, v_i$.

Usando la gráfica de la figura 1.17, si examinamos el circuito euleriano $P = A, B, C, D, A$ y queremos uno que comience en C, usando el razonamiento anterior, tendríamos: $P' = C, D, A, B, C$.

Por supuesto no todas las gráficas contienen circuitos eulerianos, algo que necesitamos garantizar al intentar encontrar un circuito euleriano que parta de un vértice v es el poder regresar a él, es decir, al menos debe tener grado dos para que una vez que comencemos el recorrido, quede al menos una arista que podamos usar para regresar. Como vimos anteriormente, una vez que tenemos un circuito euleriano que empieza en algún vértice en particular, es posible encontrar otro para cualquier vértice de la gráfica y por lo tanto esta cualidad la deben tener todos los vértices. Asimismo los vértices deben tener la cualidad de que siempre que entremos a un vértice, exista alguna forma de salir, para poder continuar el recorrido y llegar exitosamente a v; en otras palabras, deben de tener grado **par** para poder garantizar que exista una arista para llegar y otra para salir. Veremos a continuación que una gráfica conexa en la que todos sus vértices tienen grado par es euleriana; además, esta es una condición necesaria y suficiente.

Teorema 1 *Sea G una gráfica conexa. G contiene un circuito euleriano si y sólo si el grado de cada vértice en la gráfica es par.*

Demostración:

$\Leftarrow$ Supongamos primero que G contiene un circuito euleriano. Cada vez que usamos una arista para llegar a un vértice, usamos una diferente para poder salir y dado que usamos todas las aristas por tratarse de un circuito euleriano, podemos hacer pares de aristas: la que se usa para entrar y la que se usa para salir; en el caso del vértice origen, primero usamos una arista para comenzar y cada vez que regresamos, usamos una para entrar y otra para salir, mientras vamos recorriendo el circuito, en el vértice origen sólo se usan un número impar de aristas, la que se usó para salir y un número par dado por cada vez que entramos y salimos. Cuando llegamos al final del circuito, el grado del vértice origen se confirma como par porque usamos la última arista para poder regresar. De aquí, que el grado de todos los vértices es par.

$\Rightarrow$ Supongamos que el grado de cada vértice $v \in V(G)$ es par. Elegimos iniciar el circuito euleriano en $v \in V(G)$. Conforme vamos construyendo el circuito euleriano, podemos observar que cada vez que entramos a un vértice que no sea v, dado que su grado es par, siempre habrá una arista para poder salir; en el caso de v tendremos un número impar de aristas usadas, la que usamos para comenzar el recorrido más un número par por cada vez que pasemos por v, ya que usamos una arista para entrar y otra para salir, lo que garantiza que en algún momento podremos usar la arista restante para poder regresar. Cuando lleguemos de nuevo a v, si ya no quedan aristas sin usar, simplemente regresamos lo que hemos construido; si aún quedan aristas sin usar, podemos observar que lo que formamos es un circuito euleriano, pero para la subgráfica $G' = G - \varepsilon$ donde ε son las aristas que nos faltaron por usar de G. Cada vértice que tiene aristas sin usar, tiene un número par de ellas, pues como ya mencionamos cada vez que entramos a un vértice usando alguna arista, usamos otra para salir y entonces podemos seleccionar a algún $u \in V(G)$ con una arista disponible, repetir el proceso usado en v y pegar el circuito euleriano que empieza en u al que ya tenemos, repitiendo este proceso hasta que tengamos lo que queremos. Podemos garantizar que en algún momento vamos a terminar pues G tiene un número finito de aristas y vértices. De aquí que G sea

una gráfica euleriana.

■

El algoritmo de circuitos eulerianos intentará encontrar la trayectoria más larga posible partiendo desde un vértice, cuando ya no pueda continuar (cuando no haya aristas disponibles) busca alguno de los vértices por los que ya se pasó que posea una arista que no haya sido usada y repite el proceso, generándose otro circuito que se insertará en el lugar del vértice de donde salió, continuando hasta que todas las aristas se hayan usado.

Las ideas de la demostración anterior se usan en el algoritmo que describiremos a continuación:

Algoritmo de circuitos eulerianos

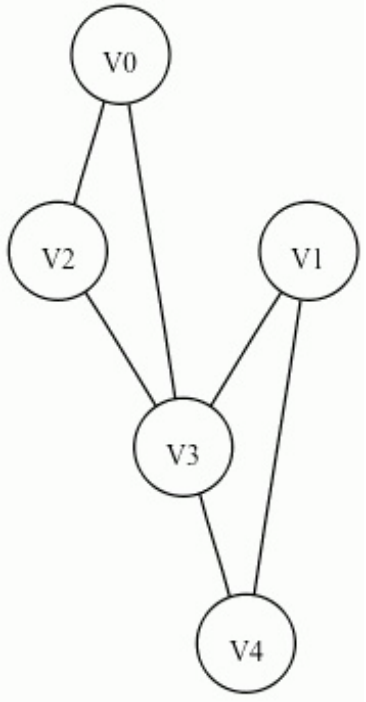
- **Objetivo:** Dada una gráfica $G = (V, E)$, tal que el grado de todos sus vértices sea par y un vértice $s \in V$, obtener el circuito euleriano de G que comienza en s.
- **Datos de entrada**: La gráfica $G = (V, E)$ y $s \in V$.
- **Salida**: C, el circuito euleriano que comienza en s.
- **Estructuras de datos:**
  1. La gráfica estará representada por listas de adyacencias.
  2. Se usará una lista C para almacenar al circuito euleriano y tendremos una copia de las aristas en el conjunto ε.
- **Método**: Se encuentra en el listado 1.1.

Listado 1.1: Algoritmo para obtener un circuito euleriano

```
1  Sea C ← [x], x ∈ V;
2  Sea E ← E;
3  Mientras (haya aristas sin usar en E)
4     Sea v un vértice arbitrario en C que tiene alguna
5        arista disponible.
6     Sea P ← [v]
7     Sea u ← v;
8     Mientras (∃ e = ux disponible)
9       E ← E − e;
10      P ← P + e + x;
11      u ← x;
12    /* Fin de ciclo: Mientras (∃ e = ux disponible) */
13    Sustituir a v por P en C;
14 /* Fin de ciclo: Mientras (haya aristas sin usar en E) */
15 C es el circuito euleriano;
```

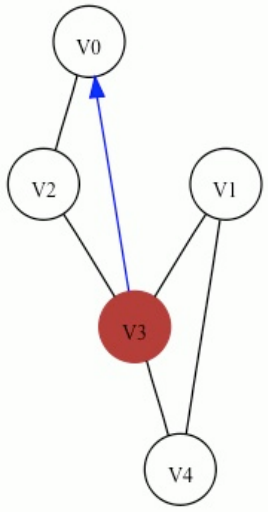
Para ilustrar el funcionamiento del algoritmo, lo ejecutaremos sobre la siguiente gráfica:

Figura 1.18: Ejemplo de una gráfica euleriana



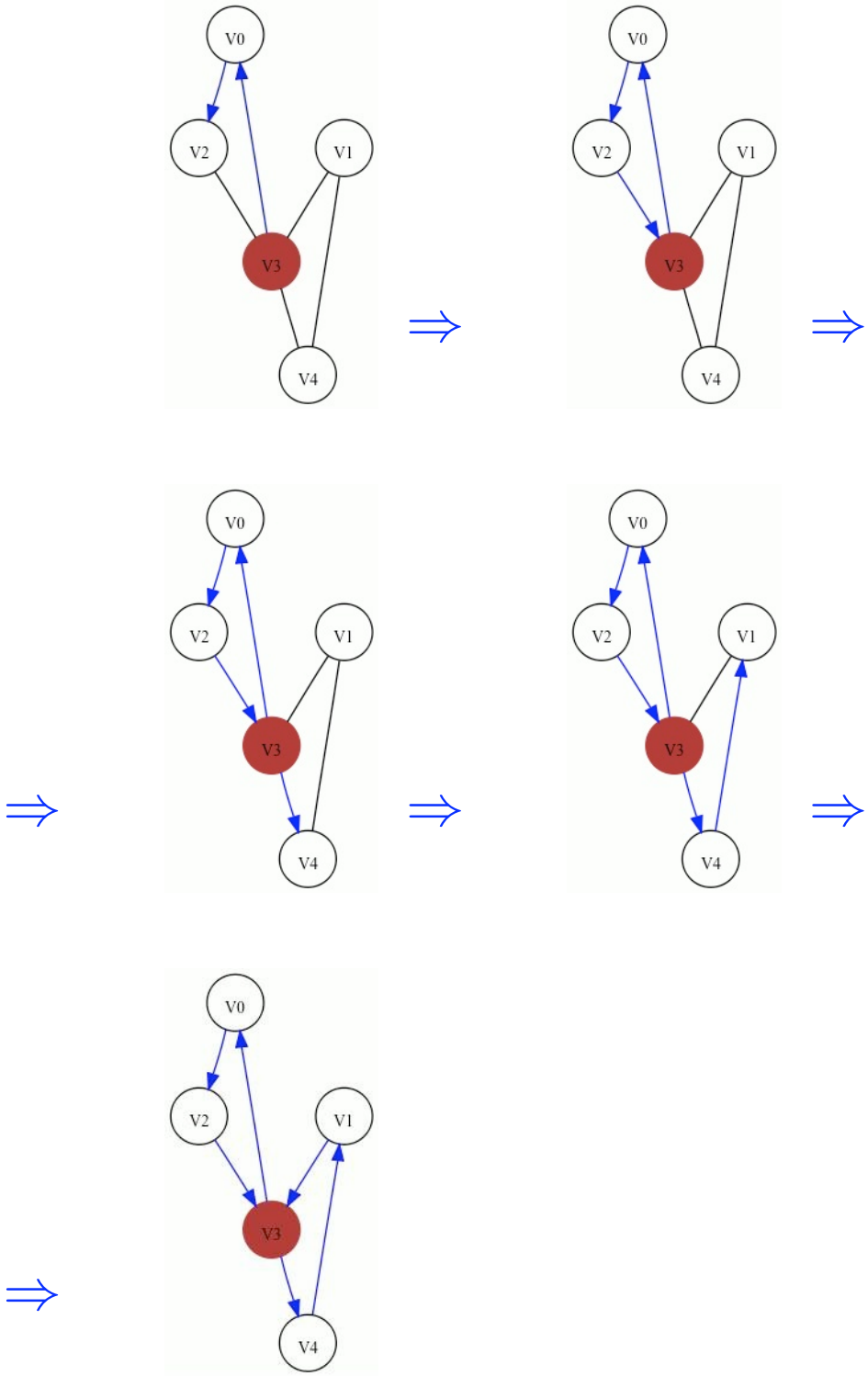
Empezaremos definiendo $\varepsilon = \{V_2V_0, V_3V_0, V_3V_2, V_3V_1, V_4V_1, V_4V_3\}$ y escogeremos $C = [V_3]$; después, como sí tenemos aristas disponibles, haremos $P = [V_3]$ y $u = V_3$; escojeremos a la arista V_3V_0 en la que sustituiremos la arista por una flecha azul para indicar la dirección que tendrá el circuito. Lo anterior se ilustra en la siguiente figura:

Figura 1.19: Se muestra una arista ya revisada dándole dirección



Para este paso, $P = [V_3, V_3V_0, V_0]$ y $u = V_0$; después volvemos a entrar al ciclo y como sí hay aristas disponibles, seleccionaremos la única arista disponible que es V_0V_2 y volveremos a darle dirección con una flecha. Los siguientes pasos son análogos al anterior pues estaremos entrando en el ciclo hasta que ya no tengamos aristas, así que sólo mostraremos imágenes donde se vea en qué orden fueron agregadas las aristas:

Figura 1.20: Orden en el que se fueron agregando las aristas



Como podemos ver, el algoritmo va agregando aristas, en el ejemplo que presentamos no salimos del ciclo que agrega aristas, aunque es posible que para otra gráfica lleguemos a un vértice sin aristas disponibles, cuando eso pasa el algoritmo selecciona alguno de los vértices que aún tenga aristas disponibles que ya se encuentre en el circuito euleriano armado hasta el momento y continúa con el mismo proceso hasta que todas las aristas hayan sido usadas.

1.5. Distancias en una gráfica

En esta sección se presentan dos algoritmos importantes para obtener las distancias entre vértices de una gráfica. Para ello comenzaremos definiendo el concepto de **distancia**, que usaremos pronto.

Sea $G = (V, E)$ una gráfica. La **distancia** entre dos vértices u y v en V denotada por $d(u, v)$, es la longitud de la trayectoria más corta entre u y v. Definimos $d(u, v) = \infty$ si no existe una trayectoria entre u y v.

Comenzaremos por revisar el algoritmo BFS.

1.5.1. Búsqueda en amplitud

Existe un algoritmo importante que sirve para obtener las distancias de algún vértice hacia todos los demás y que se le conoce como algoritmo BFS, sus siglas vienen del inglés *Breadth First Search* (búsqueda en amplitud).

El algoritmo selecciona un vértice s como origen de la exploración y comienza a inspeccionar la gráfica, obteniendo la distancia desde él a todos y cada uno de sus vértices alcanzables[4].

Los vecinos del vértice s se encuentran a distancia uno de él; los vecinos de los vecinos de s que no hayan sido explorados, se encuentran a distancia dos, y así sucesivamente es como BFS va encontrando las distancias desde s; por eso se considera que el algoritmo realiza una exploración en capas.

Es importante mencionar que conforme se van descubriendo vértices, se van marcando con sus predecesores, que corresponden a los vértices desde donde se visitaron por primera vez, por ejemplo, s es el predecesor de sus vecinos.

En los algoritmos que siguen usaremos notación similar a la usada en la programación orientada a objetos para referirnos a los distintos atributos (campos) que queremos que tenga cada arista o vértice de la gráfica. Por ejemplo, para denotar un atributo d de un vértice v, usaremos $v.d$.

Algoritmo BFS

- **Objetivo:** Dada una gráfica $G = (V, E)$ y un vértice $s \in V(G)$, encontrar $v.d$ para cada $v \in V(G)$, que corresponde a la distancia del vértice s a v. Además, registrar en la gráfica el camino que corresponde a esa distancia.
- **Datos de entrada**: La gráfica $G = (V, E)$ y $s \in V$.
- **Salida**: $\forall v \in V$, tal que haya un camino $s \rightsquigarrow v$, reportar $v.d$ y el camino-sv.

[4] Recordemos que un vértice $v \in V$ es **alcanzable** desde u si existe algún camino entre u y v.

- **Estructuras de datos:**

  1. La gráfica estará representada por listas de adyacencias.
  2. Cada vértice tendrá un atributo $v.\pi$ que indica cuál es el vértice predecesor y un atributo $v.d$ que indica cuál es la distancia a s.
  3. Se usará una cola $\mathfrak{C}$ en donde se guardarán los vértices que se van descubriendo, con el fin de revisarlos posteriormente.

- **Método**: Se encuentra en el listado 1.2.

Listado 1.2: Algoritmo BFS ($\mathsf{d(s,v)}, \forall \mathsf{v} \in \mathsf{V}$ alcanzable desde s)

```
1   /*(Inicialización:)*/
2     ∀v ∈ V : {v.π ← nulo; v.d ← ∞ }
3     s.d ← 0; s.π ← nulo
4     𝔠 ← ⟨s⟩
5   /*(Procesar al que esté al frente de la cola.)*/
6     Mientras 𝔠 ≠ ∅,
7       u ← frente(𝔠)
8       ∀v ∈ u.adyacencia:
9           v ← siguiente vértice en la lista de adyacencia de u
10          Si v.d == ∞ /* Primera vez que se llega a él */
11            v.d ← u.d + 1
12            v.π ← u
13            𝔠 ← 𝔠 + v
14          /* Fin: v.d == ∞ */
15      /* Fin de ciclo: ∀v ∈ u.adyacencia */
16      𝔠←𝔠−u /* Se saca a u de la cola */
17    /* Fin de ciclo: Mientras 𝔠 ≠ ∅ */
18    /* Se reportan los resultados */
19    ∀v ∈ V
20      /*(Reporta distancia:)*/
21      Escribe:"La distancia a v es " + v.d
22      /*(Reporta camino)*/
23      Repite:
24        Reporta v
25        v ← v.π
26      hasta que v == nulo
27    /* Fin de ciclo: ∀v ∈ V */
```

Cuando decimos que el algoritmo realiza una búsqueda en amplitud, queremos decir que en todo momento revisa todos los vértices posibles -los vecinos del vértice que está siendo inspeccionado- y que va marcando cada vértice con la distancia correspondiente, de acuerdo al momento en el que se descubre.

El algoritmo BFS es muy importante; en cuanto se descubre un vértice en la gráfica, se sabe cuál es la distancia del vértice inicial s a v y no sólo regresa las distancias sino también regresa un *árbol generador* de la gráfica. Como ya lo mencionamos, un **árbol generador** de G es una subgráfica de G que contiene a todos sus vértices y que además es árbol. Nos interesan los algoritmos que encuentran árboles generadores ya que éstos contienen la cantidad mínima de aristas necesaria para que la gráfica sea conexa y siga preservando todos los vértices.

Obsérvese además que el algoritmo descubre si la gráfica es *conexa*. Si quedan vértices cuya distancia sea ∞ al finalizar el algoritmo, quiere decir que no existe un camino entre el vértice origen y estos últimos, lo que indica que la gráfica es disconexa. Probaremos algunas propiedades que nos ayudarán para demostrar que el algoritmo BFS es correcto; para ello llamaremos **ciclo principal** al ciclo que comienza en la línea 8 y termina en la línea 13 del algoritmo.

Lema 1 *Sea $G = (V, E)$ una gráfica y sea $s \in V$. Para cada arista $uv \in E$, tenemos:*

$d(s, v) \leq d(s, u) + 1$

Demostración:

Si u es alcanzable desde s, entonces v también lo es. En este caso, la trayectoria de longitud menor de s a v, no puede ser mayor que la trayectoria mas corta $s \rightsquigarrow u$ y que se le agrega v, por lo tanto; se cumple la igualdad. Si u no es alcanzable, entonces $d(s, u) = \infty$ y la desigualdad se mantiene.

■

Lema 2 *Sea $G = (V, E)$ una gráfica. Después de aplicar el algoritmo BFS a G comenzando en algún vértice $s \in V$, para cada vértice $v \in V$ se cumple que $v.d \geq d(s, v)$.*

Demostración:

Lo haremos por inducción sobre el número de veces que se agregan vértices a la cola.

Caso base. La primera vez que se agrega un vértice a la cola, se realiza en la inicialización, por lo que $s.d = 0 = d(s, s)$ y para cada vértice $v \neq s$ tenemos que $v.d = \infty \geq d(s, v)$, por lo tanto se mantiene la afirmación.

Hipótesis de inducción. Al realizar la operación de agregar un vértice $v \in V$ a la cola $\mathfrak{C}$ por k-ésima ocasión (con $k = n - 1$), tenemos que para cada $v \in V$ se cumple $v.d \geq d(s, v)$.

Paso inductivo. Supongamos que nos encontramos revisando los vértices adyacentes a algún $u \in V$ durante alguna iteración del ciclo principal y que nos topamos con $v \in V$, el enésimo vértice agregado a $\mathfrak{C}$. Cuando se realiza una revisión de los vértices adyacentes a u, dado que u fue agregado en algún momento anterior, la hipótesis de inducción y el hecho de que su atributo d no es modificado durante esta iteración implica que $u.d \geq d(s, u)$. Por la asignación realizada en la línea 11 tenemos que:

$$v.d = u.d + 1$$

$$\geq d(s, u) + 1$$

Por lo cual $v.d \geq d(s, u) + 1$. El lema 1 aplicado a la arista uv nos da como resultado:

$d(s, v) \leq d(s, u) + 1$

Recapitulando, $v.d \geq d(s, u) + 1 \geq d(s, v)$ por lo que podemos deducir $v.d \geq d(s, v)$. El vértice v nunca se vuelve a agregar a la cola pues $v.d \neq \infty$ durante el resto de la ejecución y por lo tanto, este valor no vuelve a cambiar. De aquí concluimos como cierto el lema.

■

Antes de demostrar que el algoritmo BFS es correcto, debemos revisar con más cuidado como son procesados los elementos de la cola $\mathfrak{C}$ durante la ejecución. Demostraremos que en cualquier momento durante la ejecución, existen sólo dos valores posibles para el atributo $v.d$.

Lema 3 *Supongamos que durante la ejecución del algoritmo BFS sobre alguna gráfica $G = (V, E)$, la cola $\mathfrak{C}$ contiene los vértices $[v_1, v_2, \ldots, v_r]$ dónde v_1 está al frente de la cola y v_r se encuentra al final. Entonces $v_r.d \leq v_1.d + 1$ y $v_i.d \leq v_{i+1}.d$ para $i = 1, 2, \ldots, r-1$.*

Demostración:

Lo haremos por inducción sobre las siguientes operaciones realizadas sobre la cola:

- Sacar un vértice de la cola.
- Agregar un vértice a la cola.

Caso base. La primera operación realizada sobre la cola consiste en agregar al vértice inicial s, por lo que el lema se cumple trivialmente.

Hipótesis de inducción. Al realizar la $k-$ésima operación sobre la cola $\mathfrak{C}$ tenemos que $v_r.d \leq v_1.d + 1$ y $v_i.d \leq v_{i+1}.d$ para $i = 1, 2, \ldots, r-1$.

Paso inductivo. Es necesario demostrar que en la operación $n = k + 1$ se cumple el lema. Existen dos casos posibles para esta operación:

- **Agregar un vértice a la cola $\mathfrak{C}$.** Para este caso, cuando se agrega un vértice $v \in V$ a la cola, dicho vértice se convierte en v_{r+1} y en este momento, se están examinando los vértices adyacentes a $u = v_1$. Como u ya fue agregado a la cola en alguna operación anterior, por hipótesis de inducción tenemos que: $v_r.d \leq v_1.d + 1$; en la línea 11 del algoritmo establecemos $v.d = v_1.d + 1$, por lo que $v_{r+1}.d \leq v_1.d + 1$ y $v_r.d \leq v_{r+1}.d$. Las demás desigualdades no se ven afectadas por lo que el lema se cumple cuando se agrega un vértice a la cola
- **Sacar un vértice de la cola $\mathfrak{C}$.** Si la cola queda vacía después de sacar al vértice, el lema se cumpliría por vacuidad. Si esto no ocurre, v_2 se convierte en la cabeza de la cola (es decir el siguiente vértice a procesar). Por hipótesis de inducción tenemos que: $v_r.d \leq v_1.d + 1$ y $v_1.d \leq v_2.d$, por lo tanto, $v_r.d \leq v_1.d + 1 \leq v_2.d + 1$ y las demás desigualdades no son afectadas. De lo anterior, teniendo a v_2 como cabeza de la cola, el lema se mantiene.

■

El siguiente corolario nos muestra que el valor asignado en la línea 11 del algoritmo se va incrementando durante la ejecución.

Corolario 2 *Supongamos que los vértices v_i y v_j se agregan a la cola $\mathfrak{C}$ durante la ejecución del algoritmo BFS, y que v_i fue agregado antes de v_j. Entonces $v_i.d \leq v_j.d$ al momento de agregar a v_j.*

Demostración:

Inmediato del lema 3 y de que el atributo d de cada vértice es un número finito que se le asigna a lo más una vez durante la ejecución.

■

Ya estamos listos para demostrar que el algoritmo BFS es correcto:

Teorema 2 *Sea $G = (V, E)$ una gráfica y supongamos que el algoritmo BFS se ejecuta sobre G desde algún vértice inicial $s \in V$. Durante la ejecución, el algoritmo BFS descubre todos los vértices $v \in V$ alcanzables desde s y al final de la ejecución, $v.d = d(s, v)$ para cada vértice $v \in V$. Más aún, el algoritmo descubre una trayectoria de longitud mínima para cada vértice $v \neq s$ que es alcanzable desde s siguiendo a los predecesores $v.\pi$ de v.*

Demostración:

Lo haremos por contradicción.

Supongamos que para algunos vértices $v \in V$ ocurre que $v.d \neq d(s, v)$. De entre estos vértices, nos interesa revisar al vértice $v \in V$ que tiene la menor distancia de s a v; claramente, $v \neq s$ pues en la línea 3 se estableció $v.d = 0 = d(s, s)$. Por el lema 2 tenemos que $v.d \geq d(s, v)$. El vértice v debe de ser alcanzable desde s, pues de otro modo, ocurriría $d(s, v) \geq v.d = \infty$ y entonces tendríamos $v.d = d(s, v)$, contradiciendo nuestra elección. Sea u un vértice predecesor de v en una trayectoria más corta de s a v, entonces, $d(s, v) = d(s, u)+1$. Por la forma en que seleccionamos a v y dado que $d(s, u) < d(s, v)$, tenemos que $u.d = d(s, u)$. Recapitulando tenemos:

$$v.d > d(s, v) = d(s, u) + 1 = u.d + 1$$

Ahora consideremos el momento en el que el algoritmo BFS selecciona u de la cola $\mathfrak{C}$ en la línea 7. Durante la revisión de los vértices adyacentes a u, pueden ocurrir dos casos cuando llegamos a v:

- $\mathbf{v.d} = \infty$. En este caso, asignaremos $v.d = u.d + 1 = d(s, v)$ que no es posible por las condiciones bajo las que seleccionamos v.
- $\mathbf{v.d} \neq \infty$. Aquí, v ya fue removido de la cola y por el lema 2 tenemos $d(s, v) \leq v.d$; por el corolario 2 tendremos que $v.d \leq u.d$, de donde $d(s, v) \leq u.d$ y como vimos, $u.d = d(s, u)$; por lo cual deducimos $d(s, v) \leq d(s, u)$ y entonces $d(s, v) < d(s, u) + 1$, que contradice el lema 1.

Lo anterior demuestra que $v.d = d(s, v)$ para todo $v \in V$. Como no existe un vértice para el cual $v.d \neq d(s, v)$, todos los vértices alcanzables desde s fueron descubiertos. Para finalizar, observemos que si $v.\pi = u$, entonces $v.d = u.d + 1$. De aquí que podamos obtener una trayectoria más corta de s a v, usando primero una trayectoria más corta de s a u y agregando la arista uv.

■

Otro punto importante es que BFS regresa un árbol generador de la gráfica, lo cual demostraremos a continuación:

Teorema 3 *Sea G una gráfica conexa; las aristas usadas en BFS forman un árbol generador de G.*

Demostración:

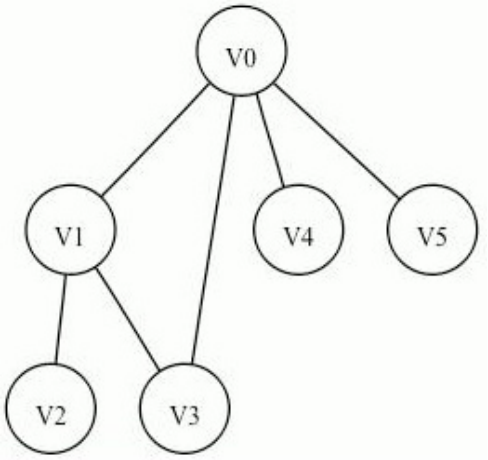
Por el teorema 2, podemos observar que después de terminar de ejecutar el algoritmo BFS se ha generado una subgráfica de G, G', y por el paso 10 observamos que nunca se agregan vértices que ya se hayan añadido evitando ciclos; además G' es conexa ya que también se demostró que se marcan los caminos más cortos del vértice inicial v a cualquier vértice u de G, si queremos mostrar un camino entre cualesquier v_i, v_j,

simplemente unimos el camino de v_i a v con el de v_j a v que encontró BFS.

■

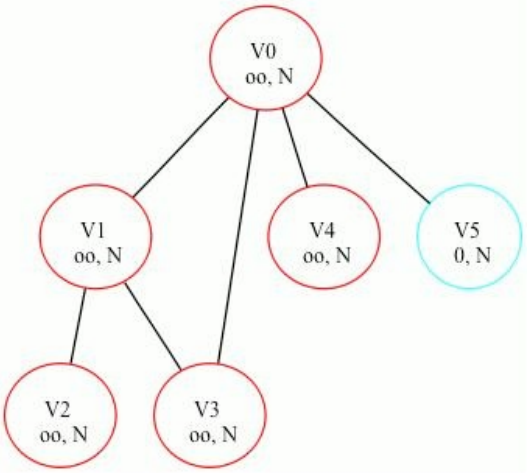
A continuación daremos un ejemplo de la ejecución de BFS, para lo cual usaremos la gráfica en la figura 1.21:

Figura 1.21: Gráfica a la que se aplicará el algoritmo BFS



Primero el algoritmo establece $v.d = \infty$ y $v.\pi = nulo$ para todo $v \in V(G)$, después seleccionamos aleatoriamente un vértice, digamos V_5, y definimos $v_5.d = 0$; además hacemos $\mathfrak{C} = [V_5]$. En la figura 1.22 este vértice está marcado más claro.

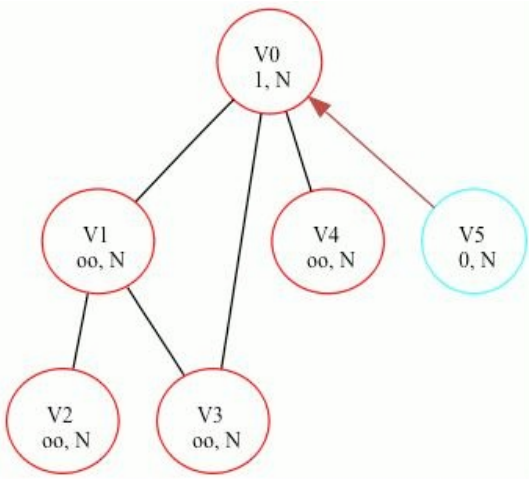
Figura 1.22: Marcando el vértice inicial



En la imagen anterior observamos que dentro de los vértices se encuentran dos valores bajo el nombre del vértice, el que está a la izquierda corresponde a la distancia al vértice origen ($v.d$) y el que se encuentra a la derecha al predecesor actual del vértice ($v.\pi$), cuando $v.\pi = \mathrm{N}$, quiere decir que aún no tiene predecesor.

Después hacemos $u = V_5$ y para su único vértice adyacente V_0 tenemos $V_0.\pi = V_5$, $V_0.d = 1$, $\mathfrak{C} = [V_5, V_0]$ y asignamos dirección a la arista como se muestra en la figura 1.23, a continuación.

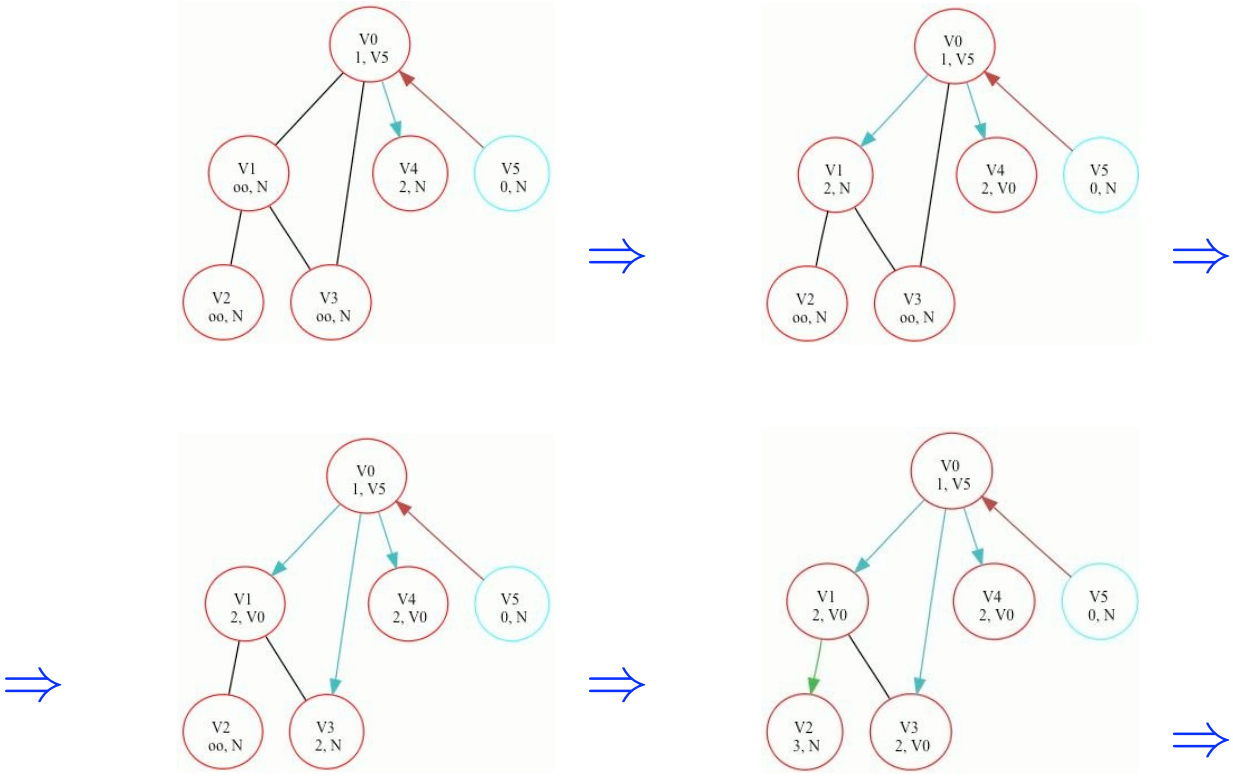
Figura 1.23: Ejecución de BFS



En las siguientes imágenes, las aristas de los vértices que se encuentren a la misma distancia del origen se marcarán del mismo color.

Los siguientes pasos son completamente análogos por lo que sólo mostraremos las imágenes correspondientes:

Figura 1.24: Orden en el que se fueron agregando las aristas (BFS)



La información obtenida por BFS -como las distancias del vértice inicial a los demás, el árbol generador o saber si una gráfica es conexa- se usa en varios problemas, como:

- Conocer si existen comunicaciones adecuadas entre varias ciudades en donde cada ciudad sería un vértice y las aristas serían las carreteras entre éstas.
- En una gráfica que represente el recorrido de algún turista por varios lugares, si la gráfica no es conexa, querría decir que hay lugares a los que no puede llegar.
- En una gráfica en la que diferentes lugares sean vértices y las aristas representen calles, BFS ayudaría a saber qué tanto podríamos tardarnos en llegar de un lugar a otro.
- El árbol generador de una gráfica puede facilitarnos encontrar caminos desde un lugar dado, pues tenemos la menor cantidad de aristas por recorrer.

El inconveniente de BFS es que sólo está hecho para gráficas en las que cada arista representa la misma "dificultad", es decir que si tenemos un vértice v_i con aristas a u_i y u_j, para BFS es igual de complicado o costoso ir a cualquiera de las dos; sin embargo, en algunos modelos es necesario definir alguna *prioridad* entre las aristas que nos ayude a decidir qué rumbo tomar. Por ejemplo, para el problema de buscar rutas entre ciudades, habrá ejemplares en donde existan dos caminos que representen cada uno una solución, pero una puede ser mejor que la otra, debido a las distancias entre cada lugar o ciudad. Para resolver esto es necesario incluir *pesos* [5] en las aristas usando alguna métrica que nos ayude a seleccionar por dónde seguir; por ejemplo, para el problema de las ciudades se puede usar el tiempo de recorrido entre ciudades como peso de las aristas, o bien, otra medida puede ser la distancia real. Para este caso, usaremos gráficas con pesos en las aristas que denotaremos con $G = (V, E; w)$. Como lo hemos venido manejando, V es el conjunto de vértices, E el conjunto de aristas y tendremos una función de peso $w : E \to \Re$ que va de las aristas en los reales[6]. Una gráfica con una función de pesos se denomina **red.** La definición de **longitud** de un camino c no cambia para estas gráficas, se define como la suma de las aristas que contiene y usaremos $|c|$ para denotarlo. El **peso** de un camino c se define como la suma de todos los costos de las aristas que lo forman y usaremos $w(c)$ para referirnos a él. La distancia de un vértice u a un vértice v la denotaremos con $d(u, v)$ con $u, v \in V$ y será igual al peso de la trayectoria de u a v que tenga menor costo en G, dicha trayectoria representa la **ruta** más corta entre el vértice u y el vértice v.

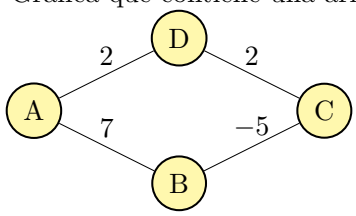
1.5.2. Algoritmo de distancias de Dijkstra

El algoritmo de distancias de Dijkstra encuentra los caminos más cortos -o de menor peso- de un vértice v a todos los alcanzables desde él; se usa en gráficas en las que las aristas tienen asignado un peso, pero éste debe ser positivo o cero, pues si existiera un ciclo tal que la suma de los pesos en sus aristas sea negativo, cada vez que recorramos el ciclo, la distancia entre v y los vértices del ciclo se irían acortando infinitamente, haciendo imposible dar una respuesta satisfactoria.

Existe otro problema al aplicar el algoritmo de distancias de Dijkstra para gráficas que contienen aristas con peso negativo. Consideremos, por ejemplo, la gráfica mostrada en la figura 1.25.

[5] Usaremos indistintamente *peso* y *costo* para referirnos al valor asignado a cada arista
[6] Nos limitaremos a los reales *positivos.*

Figura 1.25: Gráfica que contiene una arista negativa



En la gráfica anterior, si ejecutamos el algoritmo de distancias de Dijkstra que explicamos más adelante, y empezamos en A, seleccionaremos la arista AB, estableciendo $d(A, B) = 7$, después escogeremos la arista AD y tendremos que $d(A, D) = 2$ y entonces sacaremos a A de la lista de vértices por revisar. Continuando con D, seleccionando la arista DC y estableciendo $d(A, C) = 4$, sacamos a D de la lista de vértices por revisar y continuamos con C, después seleccionamos la arista CB y marcamos $d(A, B) = -1$, sacamos a C de los vértices por revisar y después revisamos a B, que ya no tiene aristas que pueda usar con lo que el algoritmo termina, sin darse cuenta de que el camino más corto de A a C es $\{A, B, C\}$ pues la distancia de dicho camino es 2.

El algoritmo de distancias de Dijkstra es bastante eficiente y es un algoritmo *glotón* o *ávido* pues en todo momento toma la decisión local más conveniente. Trabaja usando una cola de prioridades para los vértices que se van descubriendo y manteniendo en la cabeza al vértice descubierto más cercano al origen. Trabaja de manera muy similar a BFS, siendo la principal diferencia que la distancia en BFS está determinada por el número de aristas en los caminos, mientras que en el algoritmo de distancias de Dijkstra la distancia es la suma de los pesos de las aristas. Otro cambio importante es la manera en la que se selecciona al siguiente vértice para ser marcado, pues en BFS se usa una cola simple con el criterio de que el primero que llega es el primero que sale, mientras que en el algoritmo de distancias de Dijkstra, la cola es de prioridades, donde los vértices en la cola se reorganizan continuamente para que el vértice a procesar sea el que presente menor distancia al origen. Finalmente, como ahora tenemos pesos en las aristas, es necesario realizar una revisión adicional, ya que si descubrimos una trayectoria más corta entre el origen y otro vértice, el predecesor y la distancia deben ser actualizados, con el fin de que al terminar el algoritmo quede un árbol con las trayectorias más cortas entre s y los demás vértices.

A continuación detallaremos el algoritmo:

- **Objetivo.** Dada una gráfica $G = (V, E; w)$ tal que todos los pesos en las aristas son positivos o cero, y un vértice origen s, determinar $v.d$, la distancia del vértice origen a cada uno de los vértices alcanzables desde s.
- **Datos.** La gráfica G y el vértice s.
- **Salida.** El valor de $v.d$ para cada vértice alcanzable desde s.
- **Estructuras de datos:**

  1. La gráfica estará representada por listas de adyacencias.
  2. Cada vértice tendrá un atributo $v.\pi$ que indica cuál es el vértice predecesor y un atributo $v.d$ que indica cuál es la distancia actual desde s. Al finalizar el algoritmo, $v.d$ tendrá el costo de la ruta más corta de s a v.

3. Se usará una cola $\mathfrak{C}$ de prioridades en donde se guardarán los vértices que se van descubriendo, con el fin de revisarlos posteriormente.

- **Método**: Se encuentra en el listado 1.3.

Listado 1.3: Algoritmo de Dijkstra para distancias más cortas

```
1   /* Inicialización: */
2    ∀v ∈ V :
3         v.d ← ∞
4         v.π ← nulo
5    /* Ciclo: ∀v ∈ V :
6    s.d ← 0
7    C_P ← ⟨s⟩
8    /* Procesar al que esté al frente de la cola. */
9    Mientras C ≠ ∅
10        u ← v tal que v.d ≤ x.d, para toda x ∈ C
11        ∀v ∈ lista de adyacencia de u:
12             v ← u.primero; w ← w(u, v)
13             Si v.d == ∞
14                  v.d ← u.d + w
15                  v.π ← u
16                  C_P ← C_P + u
17             /* Fin: v.d == ∞ */
18             Si v.d > u.d + w
19                  v.d ← u.d + w
20                  v.π ← u
21             /* Fin: v.d > u.d + w */
22        /* Ciclo: ∀v ∈ lista de adyacencia de u*/
23        C_P ← C_P − u
24   /* Ciclo: Mientras C ≠ ∅ */
25
26   /* Reporta Resultados: */
27   ∀v ∈ V
28      /*(Reporta distancia)*/
29      Escribe "Distancia␣de␣"+ s+"␣a␣" +v+"␣es␣"+v.d
30      /*(Reporta camino)*/
31       Repite:
32          Reporta v
33          u ← u.π
34       hasta que u == nulo
35   /* Ciclo: ∀v ∈ V */
```

Antes de demostrar que el algoritmo de Dijkstra es correcto, probaremos algunas propiedades que nos ayudarán más adelante. En las siguientes demostraciones trabajaremos con una gráfica que tiene pesos positivos o cero y llamaremos **ciclo principal** al ciclo que comienza en la línea 9 del algoritmo y termina en la línea 23.

Lema 4 *(Desigualdad del triángulo). Sea $G = (V, E; w)$ una gráfica conexa en la que el vértice inicial es s. Para todas las aristas $uv \in E$ se cumple que $d(s, v) \leq d(s, u) + w(u, v)$.*

Demostración:

Si $d(s, v)$ está definida, quiere decir que existe una trayectoria t de s a v tal que $w(t) = d(s, v)$. Como $d(s, v)$ es la suma de los pesos de la trayectoria de menor peso de s a v, $w(t) \leq w(q)$ para cualquier otra trayectoria q de s a v. En particular para q una trayectoria que va de s a u, y de u a v por la arista uv, se cumple $w(t) \leq w(q) = d(s, v) + w(u, v)$ y por lo tanto $d(s, v) \leq d(s, u) + w(u, v)$.

■

Lema 5 *(Propiedad de cota superior). Sea $G = (V, E; w)$ una gráfica con pesos. Sea $s \in V$ el vértice inicial. Después de ejecutar la inicialización del algoritmo sobre G, $v.d \geq d(s, v)$ para todo $v \in V$ y esta invariante se mantiene durante toda la ejecución. Más aún, cuando $v.d = d(s, v)$, $v.d$ ya no cambiará.*

Demostración:

Haremos una demostración por inducción sobre el número de iteraciones realizadas sobre el ciclo principal.

Caso base. Para la iteración 0, $\mathfrak{C}_P = [s]$; por lo tanto tenemos que $v.d = \infty$ para todo $v \neq s$, $v.d \geq d(s, v)$ y $s.d = 0 = d(s, s)$; por lo que se cumple la invariante.

Hipótesis de inducción. Para la iteración k del ciclo, tenemos que $v.d \geq d(s, v)$ para todo $v \in V$.

Paso inductivo. Sea $e = uv$ la siguiente arista a revisar. Antes de ejecutar esta iteración, por hipótesis de inducción, $v.d \geq d(s, v')$ para todo $v' \in V$. Después de esta iteración, el único valor que puede cambiar en la línea 19 es $v.d$; de hacerlo, tendremos:

$v.d = u.d + w(u, v)$

$\geq d(s, u) + w(u, v)$ (pues por hipótesis de inducción $u.d \geq d(s, u)$)

$\geq d(s, v)$ (ya que usando la desigualdad del triángulo se verifica que $d(s, u)+w(u, v) \geq d(s, v)$)

Para ver que el valor de $v.d$ nunca cambia una vez que $v.d = d(s, v)$, hay que mencionar que no existen pasos en el ciclo en donde se incremente el valor de $v.d$ y tampoco es posible que se decremente pues como acabamos de ver, $v.d$ siempre es mayor que $d(s, v)$.

■

Lema 6 *Sea $G = (V, E, w)$ una gráfica con pesos y sea $uv \in E$. Inmediatamente después de ejecutar una iteración del ciclo principal, tenemos que $v.d \leq u.d + w(u, v)$*

Demostración:

Si justo antes de ejecutar la iteración tenemos $v.d \geq u.d + w(u, v)$, entonces haremos $v.d = u.d + w(u, v)$. Si esto no ocurre, $v.d < u.d + w(u, v)$ y además no cambia su valor al finalizar esta iteración.

■

Lema 7 *(Propiedad de convergencia). Sea $G = (V, E; w)$ una gráfica con pesos. Sea $s \in V$ el vértice inicial, $u', v \in V$ y sea $s \rightsquigarrow u' \rightarrow v$ una trayectoria tan corta como es posible en G para los vértices s, v. Después de ejecutar la inicialización del algoritmo sobre G y continuar con la ejecución, hasta que en el paso en la línea 10 del algoritmo se seleccione $u = u'$; si $u'.d = d(s, u')$ para ese momento, $v.d = d(s, v)$ después de este paso y durante toda la ejecución.*

Demostración:

Por la propiedad de cota superior, si $u'.d = d(s, u')$ en cualquier momento antes de seleccionar $u = u'$ en la línea 10, esta igualdad se mantiene durante toda la ejecución. En particular, después de ejecutar esta iteración, tendremos:

$v.d \leq u'.d + w(u', v)$ (por el lema 6)

$= d(s, u') + w(u', v)$ (ya que por hipótesis $u'.d = d(s, u')$)

$= d(s, v)$

De la propiedad de cota superior, tenemos que $v.d \geq d(s, v)$ por lo tanto concluimos $v.d = d(s, v)$.

■

Ahora demostraremos que el algoritmo nos regresa la distancia de cada vértice $v \in V$ al vértice inicial s. Para hacerlo, definiremos al conjunto $S = \{v \in V \mid v \text{ ya salió de } \mathfrak{C}_P\}$ y es necesario demostrar que $d(s, v) = v.d$ al momento de sacar v de $\mathfrak{C}_P$.

Teorema 4 *Ejecutar el algoritmo de Dijkstra en una gráfica, termina con $v.d = d(s, v)$ para todo $v \in V$.*

Demostración:

Usaremos la siguiente invariante:

Al comienzo de una iteración en el ciclo principal, se cumple que $v.d = d(s, v)$ para cada vértice v contenido en S.

Es suficiente demostrar que para cada vértice $u \in V$, al momento de ser agregado a S, se cumple que $u.d = d(s, u)$. Cuando lleguemos a esto, demostraremos que $u.d$ ya no cambia en ningún momento posterior durante la ejecución.

Inicialización

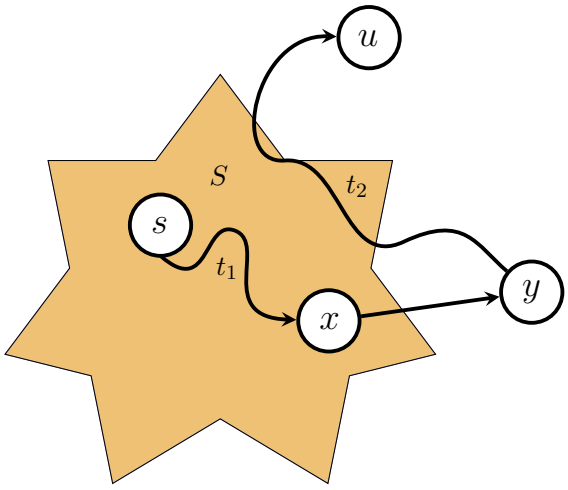
Comenzamos con $S = \varnothing$ por lo que la invariante se cumple trivialmente.

Siguientes iteraciones

Deseamos que en cada iteración, el vértice u agregado al conjunto S cumpla con $u.d = d(s, u)$.
Lo haremos por contradicción. Sea u el primer vértice agregado a S para el cual no se cumple $u.d = d(s, u)$. Nos concentraremos en revisar qué ocurre en la ejecución justo antes de agregar u a S y llegar a la contradicción de que $u.d = d(s, u)$ cuando agregamos a u, examinando la trayectoria de menor peso entre s y u.

Debemos observar que en la inicialización del algoritmo, se establece $s.d = 0$, por lo que $s.d = d(s, s)$ y por la propiedad de cota superior, dicho valor nunca cambia. Como supusimos $u.d \neq d(s, u)$, se cumple $u \neq s$ y entonces justo antes de agregar u a S, $S \neq \varnothing$. Debe existir una trayectoria de s a u ya que por la propiedad de cota superior, si $d(s, u) = \infty$, dicho valor se mantendría durante toda la ejecución, contradiciendo nuestra suposición de que $d.u \neq d(s, u)$. Como existe una trayectoria de s a u, también existe una trayectoria t de menor peso entre dichos vértices. Podemos garantizar que t tiene vértices que no están en S -pues al menos u no ha sido agregado- y vértices de S -pues demostramos que justo antes de agregar u a S, $S \neq \varnothing$-. De aquí que podamos considerar al primer vértice y contenido en t que no esté en S, y sea $x \in S$ su vértice predecesor en la trayectoria. La figura 1.26, ilustra cómo podemos descomponer la trayectoria t en $s \rightsquigarrow^{t_1} x \to y \rightsquigarrow^{t_2} u$. ($t_1$ o t_2 pueden no tener aristas).

Figura 1.26: Justo antes de agregar a u, podemos descomponer la trayectoria de menor peso t en $t_1 = s \rightsquigarrow x$ y $t_2 = y \rightsquigarrow u$, donde y es el primer vértice en la trayectoria que no está en S y $x \in S$ se encuentra justo antes de y en t. Los vértices x, y son distintos, pero es posible que $x = s$ o $y = u$. La trayectoria t_2 puede que tenga más elementos de S después de y.



Afirmamos que $y.d = d(s, y)$ cuando u se agrega a S. Para probarlo, observemos que $x \in S$. Entonces, dado que escogimos a u como primer vértice para el cual $u.d \neq d(s, u)$ cuando es agregado a S, tenemos que $x.d = d(s, x)$ cuando x fue agregado a S, por lo que la afirmación se sigue de la propiedad de convergencia.

Para concluir con la contradicción, es necesario observar que como y aparece antes de u en una trayectoria de menor peso t, $d(s, y) \leq d(s, u)$ y de aquí que:

$y.d = d(s, y)$

$\leq d(s, u)$

$\leq u.d$ (por la propiedad de cota superior)

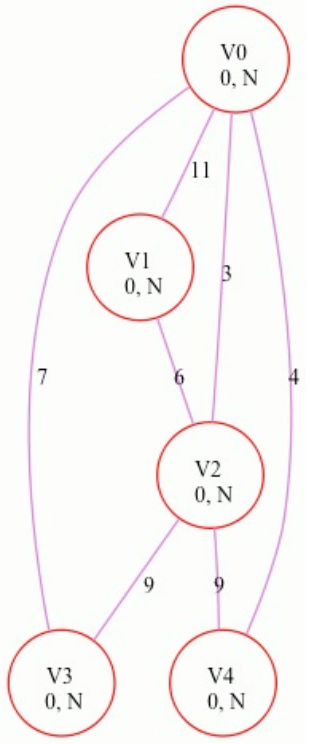
Como ambos vértices u y y no estaban en S cuando u fue seleccionado, tenemos $u.d \leq y.d$ que con la desigualdad anterior hacen que $y.d = d(s, y) = d(s, u) = u.d$ y consecuentemente, $u.d = d(s, u)$ que contradice nuestra elección de u como el primer vértice para el cual $u.d \neq d(s, u)$. Finalizamos con $u.d = d(s, u)$ cuando u es agregado a S y dicho valor se mantiene por el resto de la ejecución.

■

Con el teorema anterior queda demostrado que el algoritmo de distancias de Dijkstra calcula correctamente las distancias de un vértice particular a todos los demás en una gráfica con peso.

A continuación se dará un ejemplo del funcionamiento del algoritmo de distancias de Dijkstra. Consideremos la gráfica de la figura 1.27:

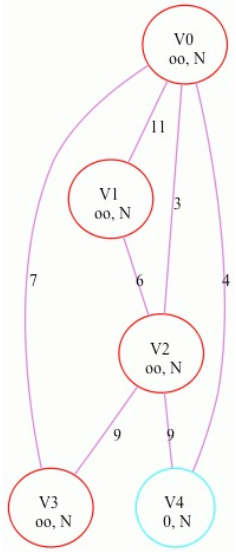
Figura 1.27: Gráfica a la que se aplicará el algoritmo de distancias de Dijkstra



Como podemos observar, la gráfica tendrá marcado en cada vértice, del lado izquierdo la distancia y el predecesor del lado derecho; como acabamos de empezar la distancia al vértice origen se establece como ∞ y el predecesor actual con una N que quiere decir que no ha sido determinado su predecesor (no ha sido alcanzado).

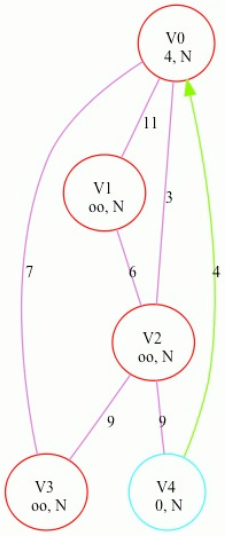
Como se indica en la línea 6 del algoritmo, seleccionamos un vértice aleatoriamente, V_4, lo marcamos para recordar que es el inicial y establecemos $V_4.d = 0$.

Figura 1.28: Marcando el vértice inicial



Posteriormente entramos al ciclo de la línea 10 pues se cumple que $\mathfrak{C}$ no es vacío; se escoje a $u = V_4$ pues es el único vértice en $\mathfrak{C}$, después se selecciona al vértice V_0 y se marca con la distancia 4 determinada por el peso de la arista, se le asigna de predecesor a V_4 y se agrega V_0 a $\mathfrak{C}$. Lo anterior se muestra en la figura 1.29:

Figura 1.29: Comenzando el algoritmo



Continuando, metemos a la cola de prioridades $\mathfrak{C}$ a V_2 y de aquí los pasos son similares al que se ilustró anteriormente. Observemos que no todas las distancias son establecidas correctamente durante el primer intento; por ejemplo, es más rápido llegar a V_2 pasando primero por V_0, por lo que el algoritmo quitará del árbol generador a la arista V_4V_2 para sustituirla por la arista V_0V_2.

Figura 1.30: Orden en el que se fueron agregando las aristas (parte 1)

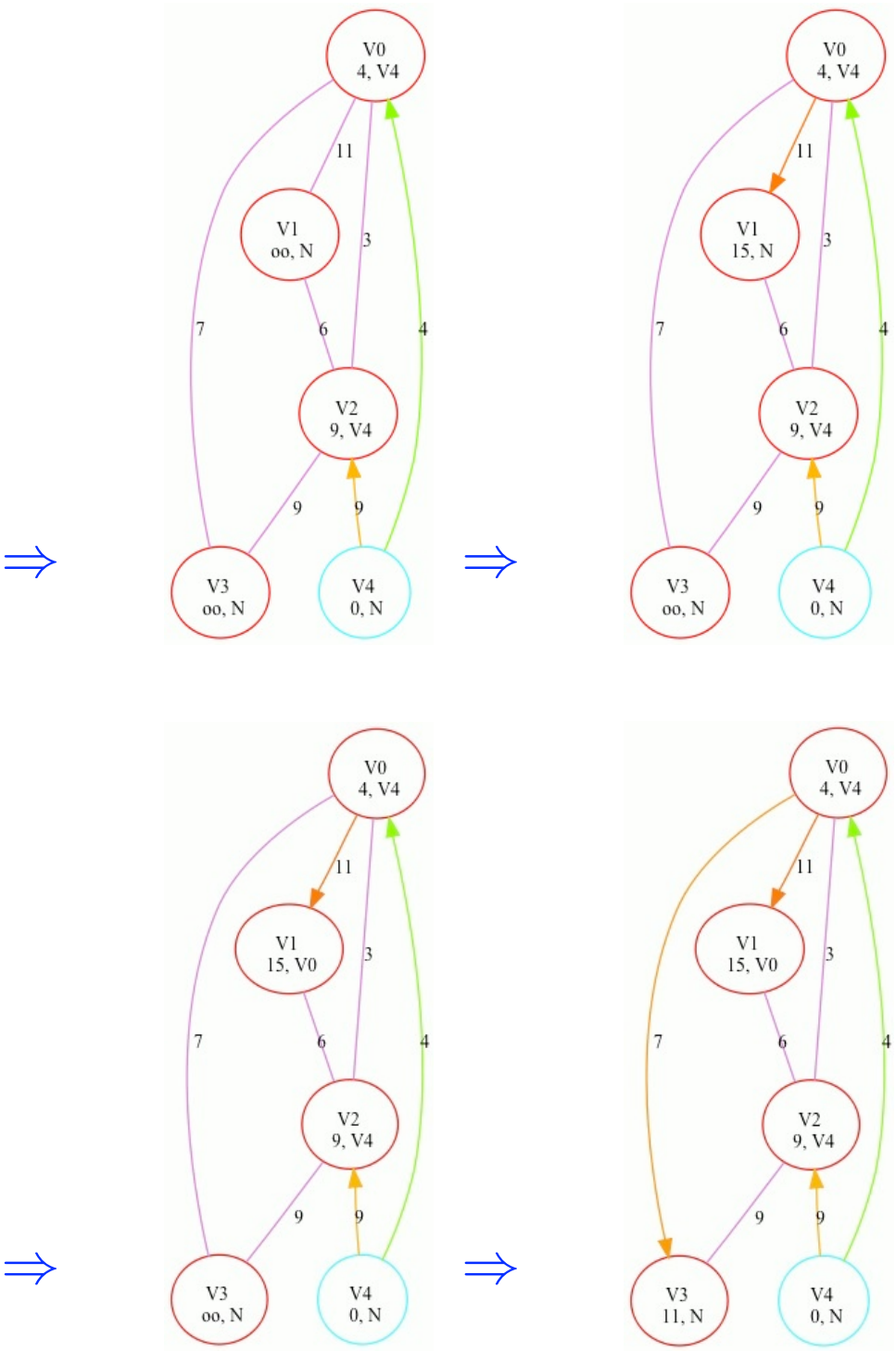
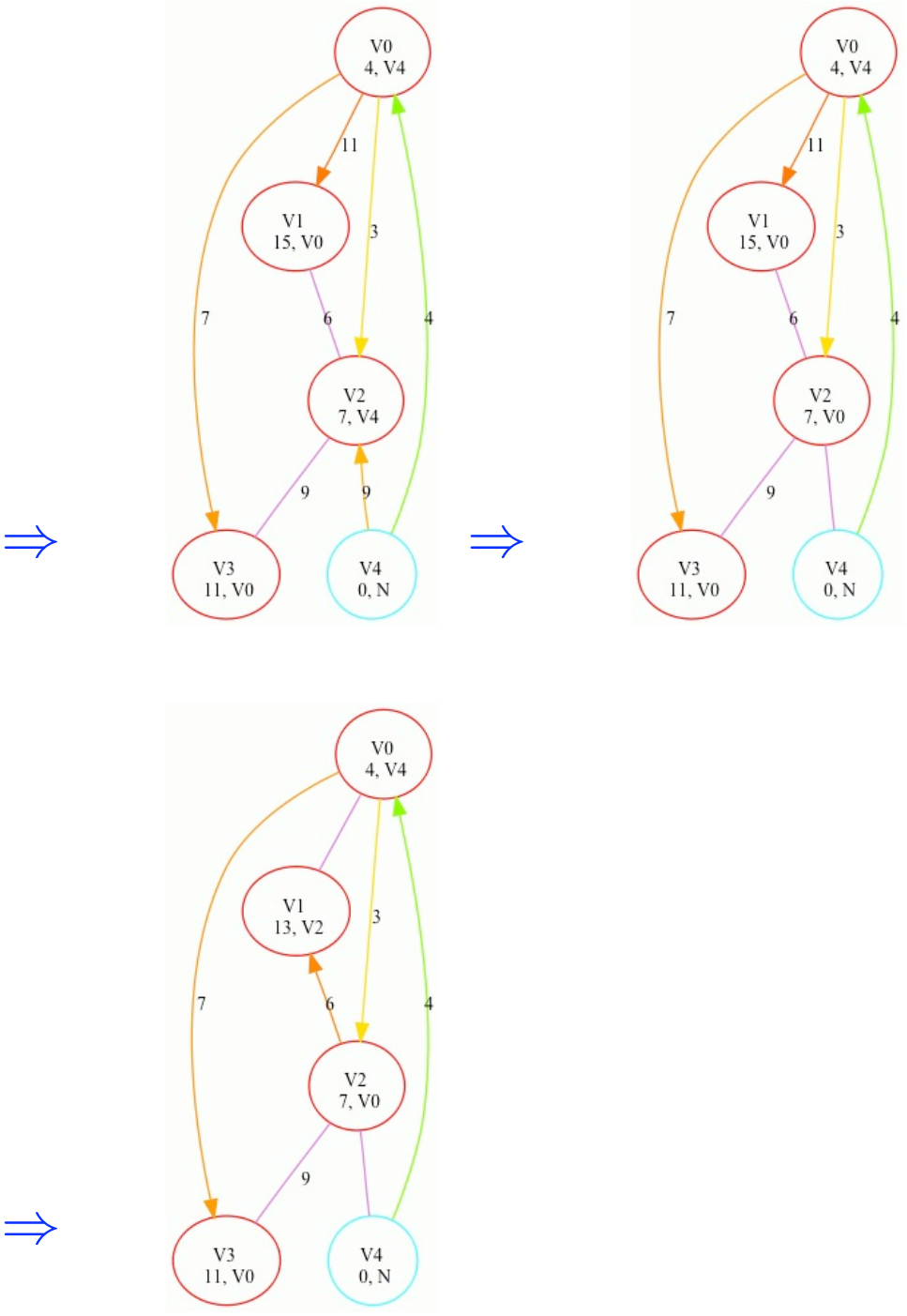


Figura 1.31: Orden en el que se fueron agregando las aristas (parte 2)



1.6. Búsqueda a profundidad

Existe otra clase de problemas que se pueden resolver mediante algoritmos de exploración de gráficas. Por ejemplo, para encontrar la salida de un laberinto, podemos sugerir la siguiente solución:

I. Tomar un vértice para la entrada del laberinto y uno para la salida.

II. Representar con un vértice cada esquina donde se pueda optar por más de un camino.

III. Representar con un vértice cada punto en el que ya no se pueda continuar.

IV. Asignar una arista a cada segmento dentro del laberinto.

Usar BFS no sería eficiente por la siguiente razón: BFS nos obliga a recorrer en cada bifurcación una única arista en la disección y regresar después a la primera. Estaríamos obligados a recorrer todas las aristas dos veces, una de ida y una de regreso; también nos obliga a revisar por niveles, obligándonos a explorar exhaustivamente la gráfica. La estrategia que nos conviene más es avanzar tanto como podamos en una sola dirección y regresar sólo si es necesario.

La estrategia de búsqueda en profundidad, conocida como DFS, por sus siglas en inglés *Depth First Search*, consiste en realizar una búsqueda lo más profundo posible hasta el momento en el que no es posible continuar; conforme se va avanzando se marcan los vértices ya visitados y se les asigna un padre de la misma forma que se asignaba en BFS, que en este caso nos serviría para no pasar por un vértice ya visitado. Una vez que nos encontremos en un vértice en donde no existan más vecinos sin visitar, regresamos al vértice padre del actual e intentamos tomar otro camino, repitiendo esto hasta que regresemos a un vértice con vecinos sin visitar o bien llegaremos a uno en el que el padre sea nulo. Con esta estrategia es posible solucionar el problema del laberinto.

A continuación detallamos el algoritmo:

- **Objetivo.** Dada una gráfica G, recorrer todos los vértices de la gráfica y dejar marcado un árbol generador de G.
- **Datos.** Una gráfica G y un vértice inicial s.
- **Salida.** El árbol generador de G.
- **Estructuras de datos.**
  1. La gráfica estará representada por listas de adyacencias.
  2. Cada vértice tendrá un atributo $v.\pi$ que indica cual es su predecesor, un atributo $v.visitado$ que indica si el vértice ya fue revisado y las aristas tendrán un atributo $xv.usada$ que indica si la arista ha sido usada.
- **Método.** Se encuentra en el listado 1.4.

Listado 1.4: Algoritmo de búsqueda a profundidad DFS

```
1   /* Inicialización: */
2       ∀v ∈ V :
3               v.visitado ← no
4       ∀e ∈ E :
5               e.usada ← no
6       x ← s
7       s.visitado ← ya
8       P ← s
9       /* Etiquetar al resto de los vértices */
10      Repite  /* Proceso de todos los vértices */
11        Mientras haya v adyacente a x sin marcar
12              v.visitado ← ya
13              P ← P⋃v
14              e ← xv
15              xv.usada ← ya
16              v.π ← x
17              x ← v
18        /*Fin del ciclo Mientras haya vértices*/
19        /*Ya no hay vértices adyacentes a x sin visitar*/
20      x ← x.π
```

Ahora demostraremos que para toda gráfica G, si se le aplica DFS, queda marcado un árbol generador. Lo haremos en dos pasos: demostrando primero que la gráfica que se va formando es conexa y con $V(G)-1$ aristas y después demostraremos que la gráfica resultante contiene a todos los vértices de G. Durante la demostración usaremos una gráfica $G' = (P, U)$ que estará formada por P, que es el conjunto de vértices ya revisados, y U, que contendrá las aristas que ya se marcaron.

Teorema 5 *Si G es una gráfica conexa y $G' = (P, U)$, en todo momento durante la ejecución de DFS se cumplen las siguientes dos afirmaciones:*

1. *G' es una gráfica conexa.*
2. *G' tiene $|V(G')| - 1$ aristas.*

Demostración:

Sea v el vértice donde se empieza a ejecutar el algoritmo; haremos una demostración por inducción sobre el número de vértices.

Caso base. Cuándo $|V(G)| = 0$, el algoritmo DFS selecciona a su único vértice, lo marca como visitado y no se realizan iteraciones pues no hay más vértices por visitar, por lo que G' es conexa y tiene $|E(G')| = |V(G')| - 1$ aristas durante toda la ejecución.

Hipótesis de inducción. Sea v_k con $k = n - 1$, el k-ésimo vértice agregado a P y $v_{k-1}v_k$ la arista marcada como usada durante esta iteración. Se cumple que G' es conexa y tiene $k - 1$ aristas.

Paso inductivo. Supongamos que G tiene al menos $n = k+1$ vértices y sea H la subgráfica de G que ha construido hasta antes de agregar el n-ésimo vértice v_n. Por la hipótesis de inducción, H es conexa y tiene exactamente $n - 1$ vértices. Por la línea 11 del algoritmo, podemos observar que v_n es adyacente a algún vértice v_i de los que ya se agregaron; de lo anterior, $G' = H \cup v_n \cup v_i v_n$, por lo que $|V(G')| = |V(H)+1| = n$

y $|E(G')| = |E(H) + 1| = n - 1$, donde, como H es conexa, los caminos entre cualquier $u_i, u_j \in V(H)$ se preservan en G' y sólo falta encontrar un camino entre v_n y $u \in V(G')$; para ello, podemos considerar los caminos de la forma $C = \{v_i, \ldots, u\}$ que existen en H y agregar v_n para obtener $C' = v_n \cup C'$. Con esto se cumple que G' es conexa y tiene $n - 1$ aristas.

■

Otra de las características importantes del algoritmo DFS cuando es aplicado a una gráfica conexa es *pasar* por todos los vértices de la gráfica. Daremos una idea de la demostración:

Teorema 6 *Si G es una gráfica conexa, el algoritmo* DFS *visita todos los vértices de G.*

Demostración:

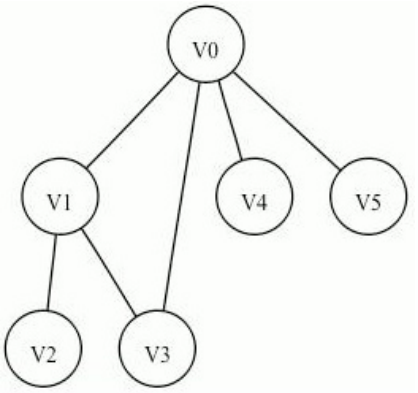
Por contradicción:

Supongamos que DFS no descubre los vértices $N = v_1, \ldots, v_n$ de G y supongamos que v es el último vértice descubierto que es adyacente a algún $v_i \in N$. Primero debemos observar que después de llegar a v, llegamos a un momento durante la ejecución en el que se visita al último vértice u que puede descubrir DFS en G, a partir de aquí, usamos los predecesores de u en búsqueda de un nuevo vértice que no haya sido visitado y en algún momento tendremos como predecesor a v, pues siempre que avanzamos vamos marcando los predecesores para poder regresar de ser necesario, por lo que pasaríamos por v de nuevo y marcaríamos a algún vértice que está en N, lo cual es una contradicción. Entonces estuvo mal suponer que DFS no descubre a todos los vértices de G.

■

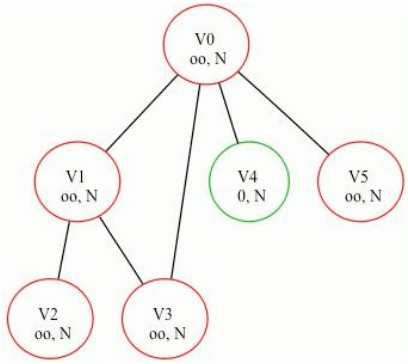
A continuación se mostrará un ejemplo de la ejecución del algoritmo, para ello consideremos la siguiente gráfica:

Figura 1.32: Gráfica sobre la que se ejecutará el algoritmo DFS



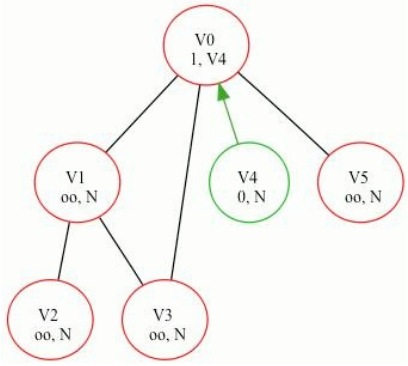
DFS seleccionará un vértice de partida, digamos V_4, y mostrará el atributo padre de todos los vértices del lado derecho con N, indicando que el vértice aún no tiene predecesor y del lado izquierdo del vértice se mostrará el orden en el que el vértice es descubierto, que para la inicialización lo pusimos como ∞. Después, en el siguiente paso, el orden en el que fue descubierto V_4 será 0. Lo anterior se muestra en la figura 1.33.

Figura 1.33: Selección del vértice inicial



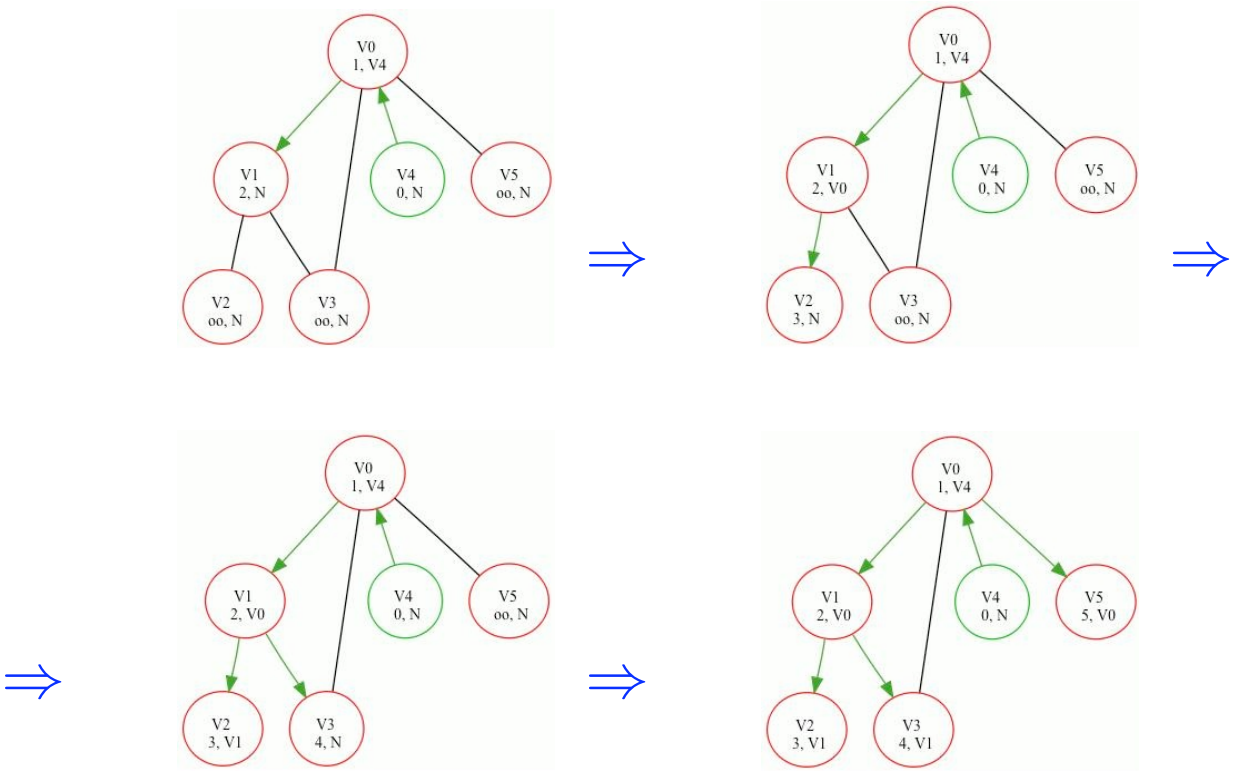
A continuación $x = V_0$, se agrega dirección a la arista V_4V_0, se escribe 1 a la izquierda del vértice V_0 indicando que fue el primero en ser descubierto y V_4 como predecesor de V_0; esto lo podemos ver en la figura 1.34.

Figura 1.34: Comenzando el algoritmo



Continuamos de la misma forma, se hace $x = V_1$, se escribe 2 a la izquierda de V_1, se pone a V_4 como predecesor, se agrega dirección a la arista V_4V_0. De aquí en adelante sólo se mostraran las imágenes que nos enseñan el orden en el que se fueron agregando aristas.

Figura 1.35: Orden en el que se fueron agregando las aristas (DFS)



Habiendo expuesto los algoritmos principales con los que trabajaremos, pasamos a describir el uso del paquete.

Parte II

Manual de usuario

Capítulo 2

Manual de usuario

2.1. Cómo instalar y ejecutar Pantera

El programa viene listo para usarse desde el momento en el que se descomprime el archivo.

Para ejecutarlo es necesario abrir una terminal y dirigirnos a la carpeta raíz del programa, y escribir

java -jar Pantera.jar

En OS X y Windows es posible abrir el programa dando doble clic al archivo Pantera.jar.

Requisitos del sistema:

- El sistema usa graphviz para la parte gráfica del seguimiento del algoritmo y para la construcción de gráficas. Es necesario tenerlo instalado; si no, el programa no funcionará. En esta liga vienen las diferentes versiones de graphviz, seleccionamos la adecuada para nuestro sistema operativo y la instalamos:

  `http://www.graphviz.org/Download.php`

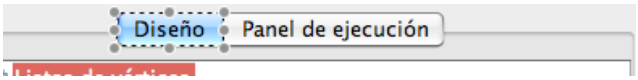
  En el sistema operativo ubuntu, que actualmente es la distribución más popular de linux, es posible instalar graphviz ejecutando el comando: sudo apt-get install graphviz.

- El sistema está desarrollado completamente en Java, específicamente se requiere de la versión 1.5 en adelante.

2.2. Diseño de gráficas

Lo primero que vamos a revisar es cómo registrar gráficas en el sistema. Para crear la gráfica con la que se va a trabajar, debemos situarnos en la pestaña de diseño dando clic en la zona que se ilustra en la figura 2.1:

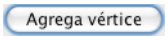
Figura 2.1: Pestaña de diseño



Veremos que la pantalla está dividida en dos partes, la primera contendrá las listas de adyacencias de los vértices y la segunda mostrará una imagen de la gráfica actual. Para el caso de las listas de adyacencias, cada lista tendrá una casilla a la izquierda, como se muestra en la siguiente figura:

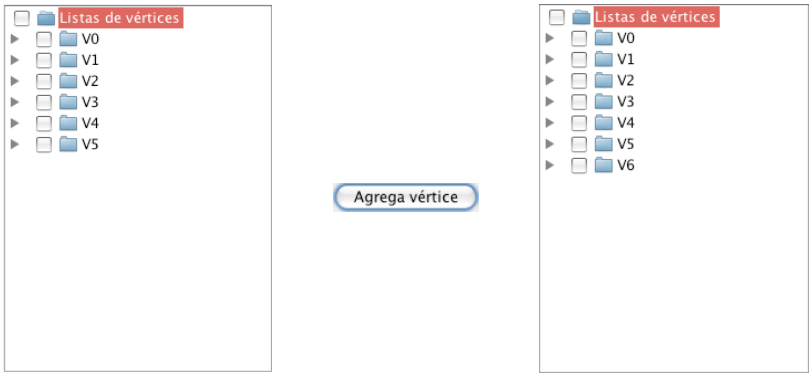
Figura 2.2: Casilla de la lista de vértices

Agregar vértices

Para agregar un vértice debemos presionar el botón 

situado bajo la lista de adyacencias; esto agregará una nueva lista de adyacencia como se muestra en la figura 2.3:

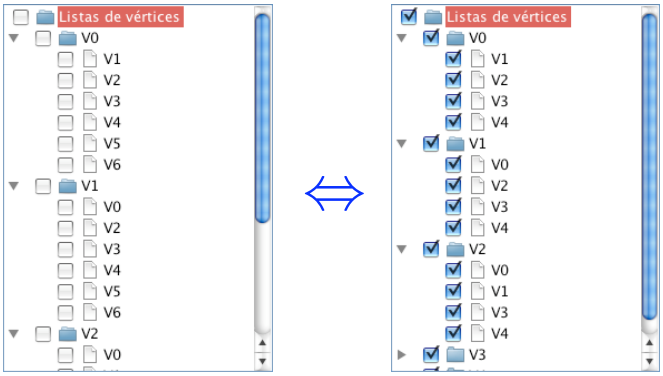
Figura 2.3: Agregar un vértice



La casilla raíz

Sin importar cuántas casillas se han marcado, excepto cuando todas están marcadas, dando clic en la casilla raíz se provocará que todas las casillas se marquen, es decir se tendrá una gráfica completa con el número de vértices que se tienen actualmente, como se muestra en la figura 2.4:

Figura 2.4: Generar gráficas completas



Cuando tengamos todas las casillas seleccionadas, dar clic en la casilla raíz provocará que todas las casillas queden *desactivadas*[1], teniendo así una gráfica sin aristas.

Agregar la lista de adyacencias de un vértice

Es posible seleccionar de un sólo clic toda la lista de adyacencias de un vértice dando clic en la casilla al lado del vértice, como se muestra en la figura 2.5:

Figura 2.5: Agrega la lista de adyacencias de $V0$

Cuando en la lista de adyacencias de un vértice se encuentran todas las casillas seleccionadas, dar clic de nuevo causará que se desactiven todas las casillas de los vértices en la lista, quedando una gráfica sin aristas; esta es una manera de "restaurar" las aristas, si se desea comenzar de nuevo. En pocas palabras, en la lista de adyacencias de cada vértice se encontrarán todos los posibles vértices con los que se le puede relacionar, cuando una casilla esté activada quiere decir que la arista pertenece a la gráfica.

Agregar una arista

Supongamos que tenemos dos vértices, i, j, y que deseamos añadir una arista entre ellos; para conseguirlo, debemos ir a la lista de adyacencias de alguno y marcar la casilla a la izquierda del otro. Por ejemplo, para colocar una arista entre el vértice V_0 y V_2; nos podemos dirigir a la lista de adyacencias de V_0 y marcar a V_2, el programa automáticamente marcará en la lista de V_2 al vértice V_0. Lo anterior se ilustra con la figura 2.6:

[1] Con casilla *desactivada*, queremos decir que una casilla no está seleccionada

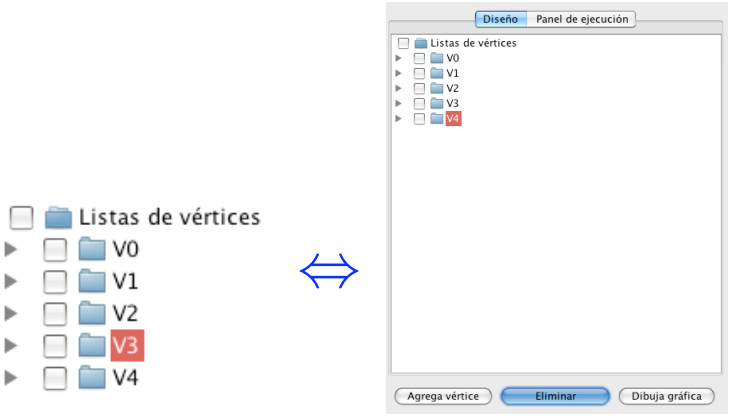
Figura 2.6: Agregar una arista

Eliminar vértices

Para eliminar vértices es necesario tener seleccionados los vértices que deseamos borrar. Esto se consigue dando un clic sobre el texto del vértice que deseamos eliminar y después damos clic en el botón 

, como se muestra en la figura 2.7:

Figura 2.7: Eliminar un vértice



Cabe mencionar que se realizarán cambios en los nombres de los vértices restantes y se hará de la siguiente manera:

- Si el número del vértice es menor al del vértice que se eliminó, quedará igual.
- Si es mayor, se le asignará la etiqueta $V + (i - 1)$, donde i es el número del vértice en cuestión.

En otras palabras, los vértices se reenumeran (renombran) con índices consecutivos en el orden en el que se encuentran.

Eliminar varios vértices a la vez

Es posible eliminar varios vértices a la vez; basta con mantener pulsada la tecla 

mientras se da clic en los vértices que se desean eliminar y después usar el botón eliminar, como se muestra en la figura 2.8:

Figura 2.8: Eliminar varios vértices

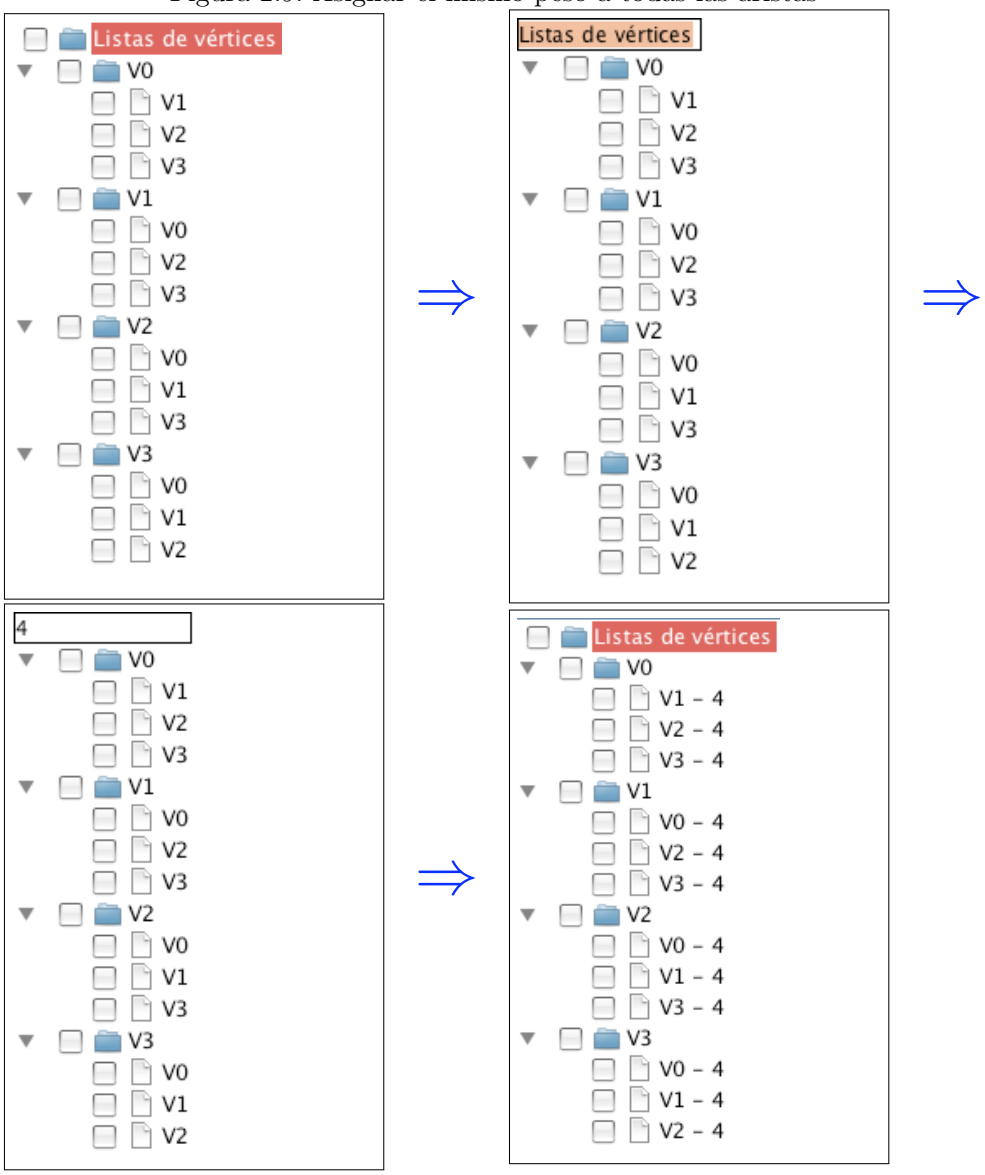
2.3. Pesos

A continuación se mencionan las diferentes formas de agregar pesos a las aristas.

Peso para todas las aristas

Deseamos ahora asignar peso a las aristas. Un primer paso es asignar un peso uniforme para toda la gráfica. Para ello nos situamos sobre la raíz del árbol y damos clic sobre la leyenda Listas de vértices; después, de la misma forma que se cambia el nombre a un archivo, damos otro clic y veremos como nos aparece un espacio donde podemos escribir, donde indicamos el peso deseado para todas las aristas y presionamos la tecla Enter para finalizar el proceso. En la figura 2.9

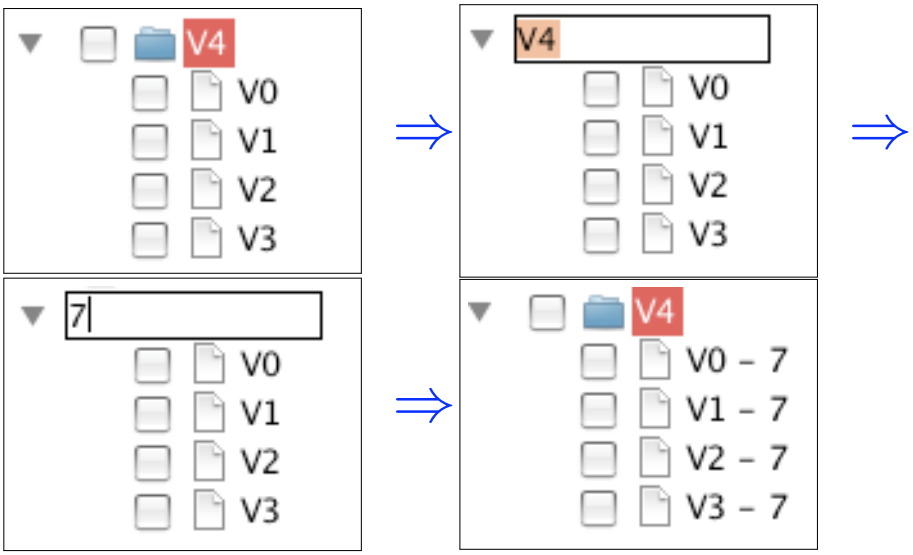
Figura 2.9: Asignar el mismo peso a todas las aristas



Asignar peso para todas las aristas de un vértice particular

Ahora lo que deseamos hacer es asignar el mismo peso a todas las aristas incidentes en un mismo vértice. Para ello nos situamos sobre el vértice, damos clic para seleccionarlo y a continuación otro clic, para que se abra el campo donde introducimos el peso; lo escribimos y presionamos la tecla enter. Las imágenes a continuación nos muestran el proceso:

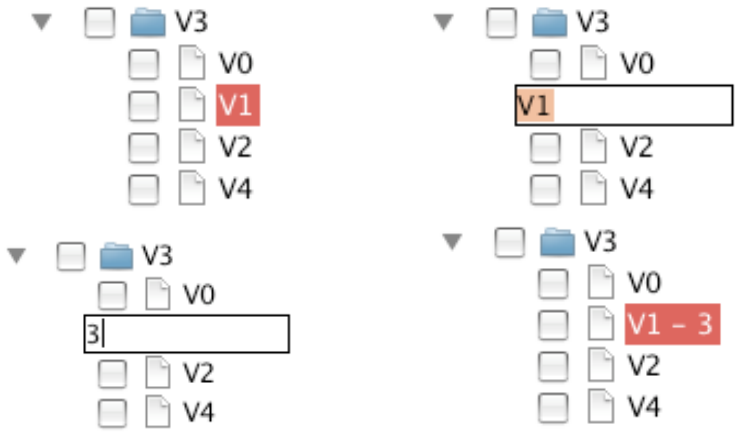
Figura 2.10: Asignar peso para todas las aristas incidentes en un vértice



Asignar un peso a una arista

Análogamente a las anteriores, si deseamos agregar peso a la arista uv debemos situarnos sobre la lista de adyacencias de u y buscar v, darle un clic para marcarlo y después otro para que aparezca el campo donde se introduce el peso deseado para la arista. En las siguientes imágenes se muestra el proceso para asignarle peso 3 a la arista V3V1:

Figura 2.11: Asignar peso a una arista

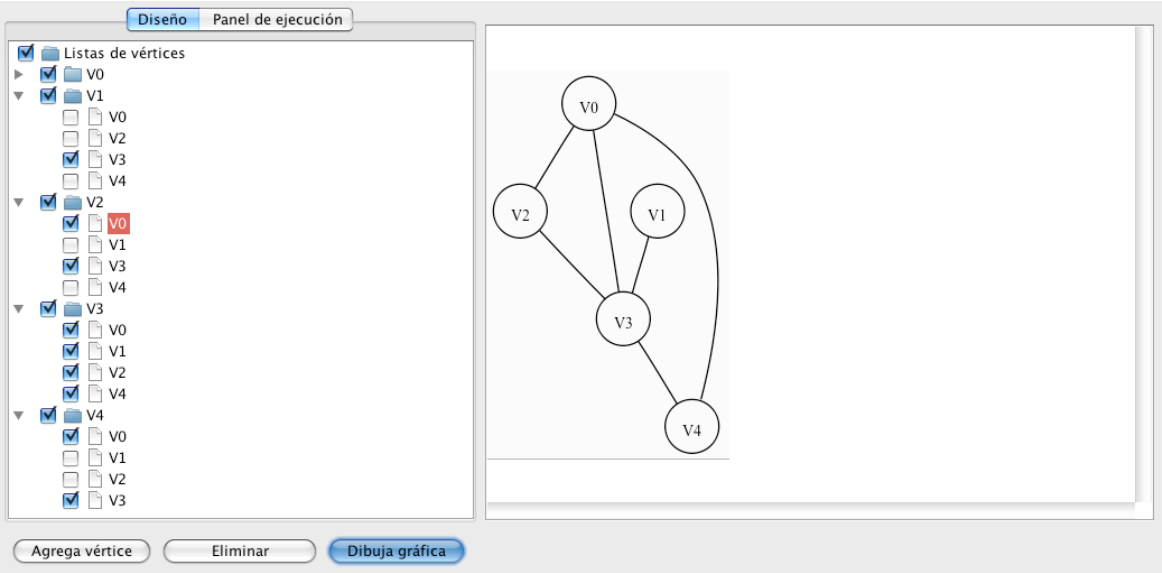


2.4. Visualización y manipulación de gráficas

Ver una imagen de la gráfica actual

Para ver una imagen de la gráfica actual, basta con dar clic en el botón Dibuja gráfica, situado bajo las listas de adyacencias. La figura 2.12 ilustra lo mencionado.

Figura 2.12: Visualizar imágenes de la gráfica diseñada



Seleccionar el vértice inicial

Es posible seleccionar el vértice inicial escribiéndolo en el campo de texto bajo la imagen de la gráfica como se muestra en la figura:

Figura 2.13: Seleccionar el vértice inicial

Si escribimos un número mayor al existente de vértices o escribimos algo que no sea un número, se seleccionará un vértice aleatorio de la gráfica.

Guardar la gráfica en el disco duro

Para guardar la gráfica actual en el lenguaje de Graphviz en el disco duro, es necesario escribir algún nombre dentro de la caja de texto, y dar un clic al botón G, como se muestra en la figura 2.14:

Figura 2.14: Guardar la gráfica actual

Una observación importante al respecto es que de la gráfica se guardan los pesos de las aristas, la aristas y los vértices; si al momento de guardarla algún vértice estaba coloreado esa información no será escrita, si se consiguió guardar la gráfica bien, esta se guardará en la carpeta Graficas en la raíz de donde se encuentra el programa y se mandará un aviso al usuario de que la gráfica fue guardada exitosamente.

Además de la gráfica, se guarda un objeto de la clase Grafica en la carpeta ObjetosGP. Si se desea volver a utilizar la gráfica guardada con el programa, es necesario no borrar este archivo de la carpeta.

Cargar una gráfica del disco duro

Para cargar una gráfica, es necesario escribir en la caja de texto el nombre de alguna que hayamos guardado previamente y dar clic en el botón C como se muestra en la figura 2.15:

Figura 2.15: Cargar una gráfica del disco duro

Al ser cargada se enviará un mensaje de éxito al usuario y se pintará la gráfica en el panel. Si ocurrió un problema al cargarla también se le avisará al usuario.

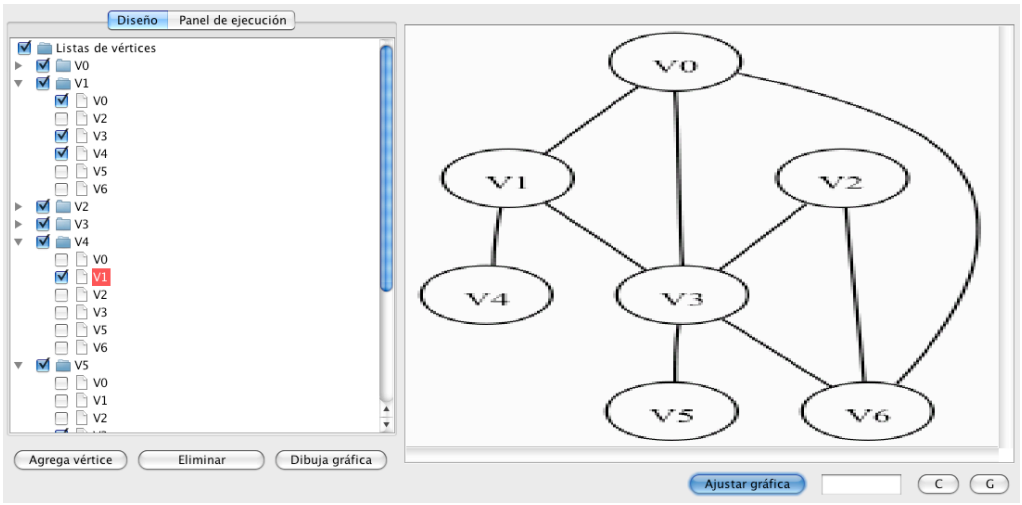
Recordemos que la información de la gráfica se encuentra en un archivo dentro de la carpeta ObjetosGP, si el archivo que intentamos cargar no existe, la gráfica no se podrá cargar.

Ajustar la gráfica al panel

Es posible ajustar la imagen de la gráfica al tamaño del panel, dando clic en el botón 

como se muestra en la figura 2.16

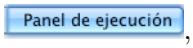
Figura 2.16: Ajustar gráfica



Capítulo 3

Ejecución de algoritmos

En esta primera versión de Pantera se han implementado las ejecuciones de los algoritmos de circuito euleriano, BFS, DFS y de distancias de Dijkstra.

Una vez creada la gráfica, damos clic en la pestaña 

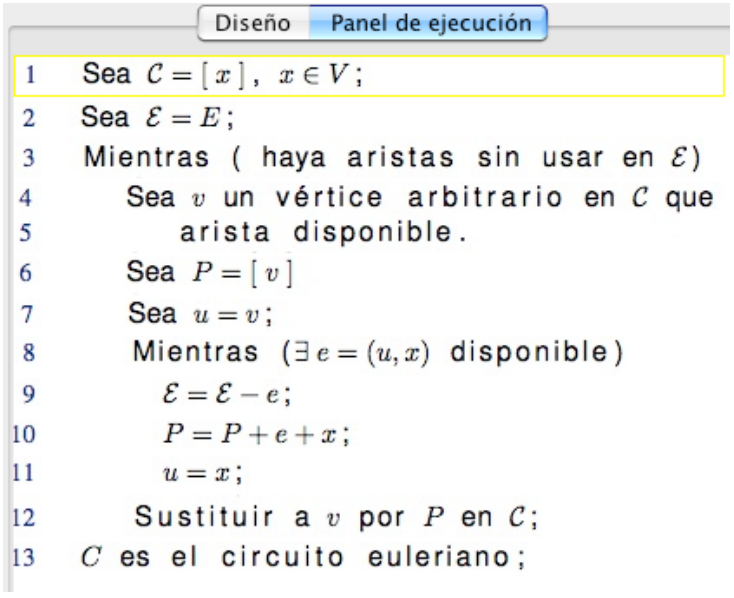
, ya que es donde podremos ejecutar los algoritmos revisados en capítulos anteriores. Su ejecución es bastante sencilla. A continuación mostramos algunos detalles de la ejecución de cada uno de los algoritmos con la intención de disipar posibles dudas al respecto.

3.1. Básicos

El panel de ejecución contiene lo siguiente:

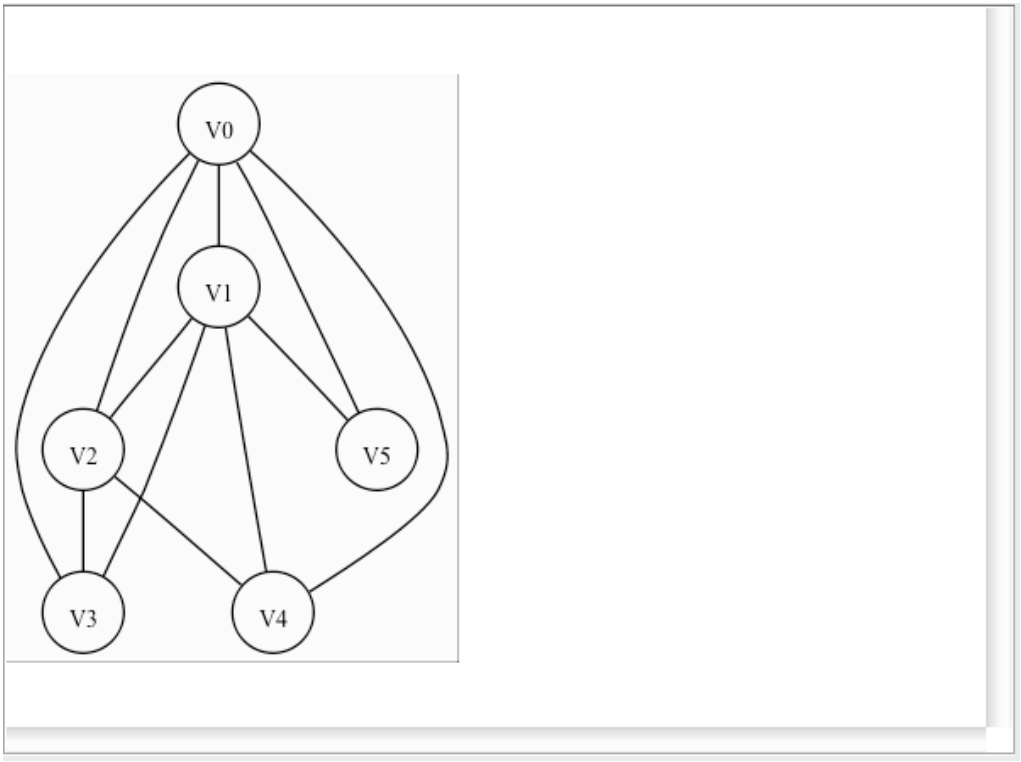
I. **Un área dónde está el algoritmo, en pseudocódigo.** En esta área se marcará con un borde amarillo el paso que se está ejecutando.

Figura 3.1: Algoritmo en pseudo-código



II. **Un espacio con el dibujo de la gráfica.** Es el mismo que se puede observar cuando se está diseñando la gráfica, pero ahora las aristas y vértices se marcarán dependiendo del estado en el que se encuentre el algoritmo.

Figura 3.2: Imagen de la gráfica

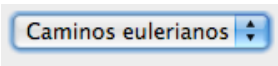


III. **Un área de texto donde se irán anotando sucesos importantes relacionados con la ejecución del algoritmo.** Aquí se almacenará un resumen de lo ocurrido con el algoritmo

Figura 3.3: Descripción de los pasos

IV. **Un selector de algoritmo a ejecutar.** Aquí se puede seleccionar cuál algoritmo se desea ejecutar.

Figura 3.4: Seleccionador del algoritmo



V. **Un botón para ir al siguiente paso del algoritmo.** Cada vez que se presiona, se avanza al siguiente paso del algoritmo. Si aún no se han ejecutado pasos, la leyenda del botón será "Comienza algoritmo".

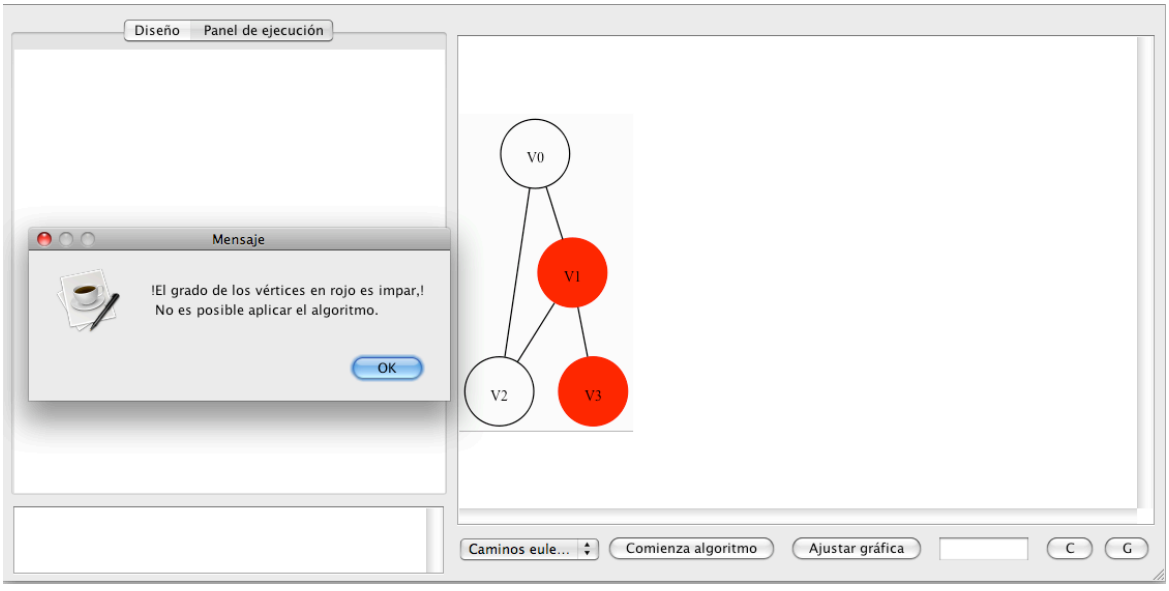
Figura 3.5: Botón para empezar

A continuación indicaremos algunas peculiaridades que puede presentar el sistema y que dependen del algoritmo particular que se esté ejecutando.

3.2. Circuitos eulerianos

Como ya mencionamos antes, el algoritmo para circuitos eulerianos elige que la gráfica con la que se va a trabajar sea euleriana. Por lo tanto, cuando intentamos ejecutar el algoritmo con una gráfica que tenga vértices de grado impar, al seleccionarlo, nos envía un mensaje de que hay vértices con grado impar, como se muestra en la figura 3.6.

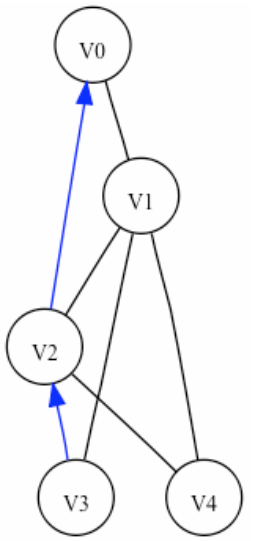
Figura 3.6: Error en el grado de los vértices



Después de dar clic al mensaje, el programa nos regresa al área de diseño, ahí debemos hacer que todos los vértices tengan grado par para poder ejecutar el algoritmo.

Una vez que tengamos una gráfica válida, conforme se vaya ejecutando el algoritmo, se irán marcando aristas dirigidas de color azul, que indican el camino euleriano que se va formando; lo anterior se ilustra en la siguiente figura.

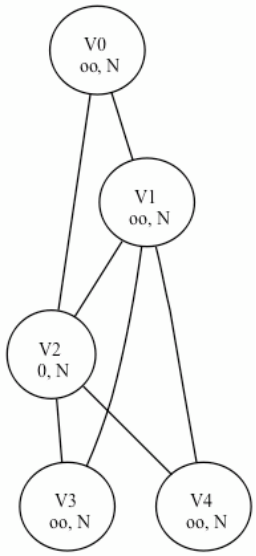
Figura 3.7: Las aristas azules que van marcando el ciclo



3.3. BFS

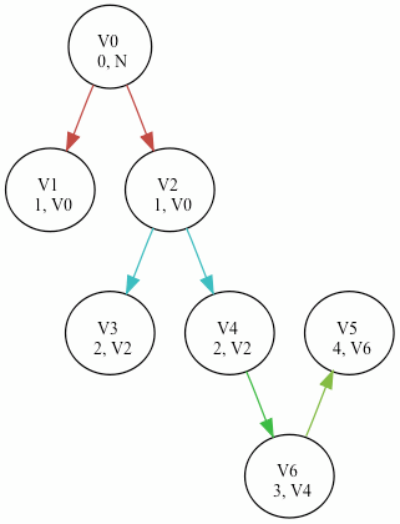
Al ejecutar este algoritmo, los vértices tendrán la distancia al vértice inicial y a su predecesor debajo del nombre. Por ejemplo, en la siguiente figura, como el algoritmo apenas va a empezar, todas las etiquetas están en infinito y los padres en nulo, denotado con la letra *N*.

Figura 3.8: Como lucen los vértices al ejecutar BFS



Asimismo, conforme el alagoritmo avanza, las aristas por las que se pasa van adquiriendo dirección y se marcan de colores. Si tienen el mismo color quiere decir que tienen la misma distancia al vértice origen, es decir que se van marcando por capas de distancias, como lo vemos en la figura 3.9.

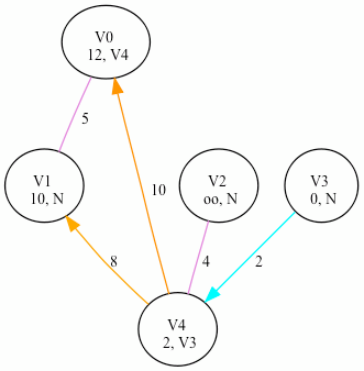
Figura 3.9: Colores y direcciones en las aristas al ejecutar BFS



3.4. Distancias de Dijkstra

Para el algoritmo de distancias de Dijkstra, lo único que cambia en la ilustración de la gráfica respecto a BFS, es que ahora se marcan los pesos en las aristas. Lo anterior se ilustra en la siguiente figura:

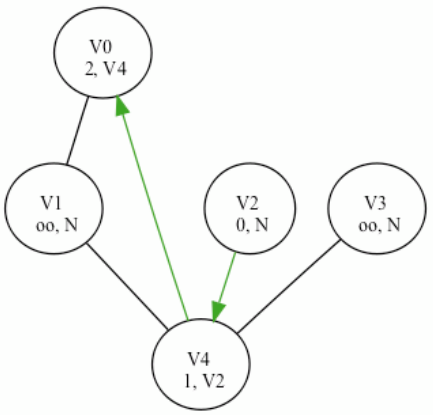
Figura 3.10: Ejecución de Dijkstra



3.5. DFS

Al ejecutar el algoritmo DFS, lo que se va marcando en los vértices son el padre (por ejemplo en el caso de V_0, su padre es V_4) y el momento en el que es descubierto (en el caso de V_0, tiene un 2 a la izquierda del padre, que indica que es el segundo vértice descubierto), como se muestra a continuación:

Figura 3.11: Ejecución de DFS



Como pudimos observar, aunque una gráfica parezca la misma, el significado de las etiquetas de los vértices depende del algoritmo que se esté ejecutando.

Por otra parte, decidimos usar aristas dirigidas para facilitar el entendimiento de lo que ocurre en el algoritmo.

Parte III

Manual del programador

Capítulo 4

Manual del programador

En este capítulo se revisarán los detalles de la implementación, solución a problemas presentados a lo largo del desarrollo y formas de extender el programa.

Consideraciones para la implementación

Antes de empezar el desarrollo, revisamos detalles relacionados con el perfil de los usuarios a los que está dirigido el programa, que son alumnos de matemáticas discretas a los cuales no les podemos pedir una interacción muy complicada con el programa; otro objetivo es el de la extensibilidad, pues se desea que se puedan agregar algoritmos de tal forma que los programadores se concentren en la implementación de éstos y no en el demás código relacionado con cómo presentar la información.

Con base en las características anteriores, y sin ningún orden particular, describimos a continuación los objetivos que cumple el programa y la forma en la que se resolvieron:

1. Las gráficas deben ser mostradas con alguna imagen que represente su estado actual. Para ello, La imagen que se muestra, se decidió que debería ser la gráfica detallando algunos cambios y fue creada usando Graphviz. Más adelante discutiremos con más detalle este paquete.

2. La interfaz con el usuario debe tener una forma amigable de crear nuestras propias gráficas. De aquí que en la interfaz del usuario exista un botón para agregar vértices y casillas para agregar o quitar aristas. Para resolver el problema de generar la gráfica, se utilizó la biblioteca JTree que representa gráficamente las listas de adyacencias de la gráfica.

3. La ejecución del algoritmo debe ser detallada y debe existir una bitácora con un resumen del proceso. Para resolver este objetivo, la ejecución del algoritmo se presenta a través de los siguientes componentes:

   - Un panel con el algoritmo actual en pseudo-código donde siempre se tiene marcado el paso actual.
   - Un cuadro pequeño con la bitácora de lo acontecido durante el algoritmo.
   - Una imagen de la gráfica que puede contener los predecesores, peso de las aristas o lo que sea necesario para mostrar el estado el algoritmo.

4. Aunque cubrimos los algoritmos más importantes en un curso introductorio de teoría de gráficas, pensamos que se deben poder agregar algoritmos, pues si se tiene una buena interfaz, se puede aprovechar al máximo. Para lo anterior se debe crear una clase que herede de Algoritmo, clase que ya contiene varios métodos útiles para mostrar el desarrollo del mismo.

5. Dado que los algoritmos se presentan a los alumnos en pseudo-código, se debe poder conocer en qué paso se encuentra la ejecución del mismo, por tal motivo se distingue el paso actual en un marco amarillo.

Soluciones a retos de implementación

A continuación listamos las diversas soluciones que se dieron a los retos durante el proceso de desarrollo del sistema:

1. Para generar la gráfica, se pensó en opciones antes de decidir la actual, desde usar complementos gráficos con los que se pueden agregar vértices y aristas mediante el uso del ratón hasta usar una matriz que representara las adyacencias de la gráfica. Se optó por el actual, porque parece el más eficiente e intuitivo de todos, pues inclusive el programa que lo hacía mediante el uso de un complemento gráfico resultaría complicado de usar si se tienen muchas aristas o muchos vértices en la gráfica.
2. Se extendió la clase JTree para manejar el trazado y eventos de la estructura del para que funcione como se menciona en el capítulo de generación de gráficas. Se tuvo que añadir la capacidad de activar o desactivar casillas de acuerdo a las acciones del usuario, por lo que se tuvieron que añadir nuevas estructuras que heredaran de las que lo manejan.
3. Desarrollar una forma de agregar algoritmos que no fuera tan complicada. La solución a este reto se detalla a continuación.

4.1. Estructura de clases y cómo agregar un nuevo algoritmo

El programa está dividido en los siguientes paquetes:

1. **Algoritmo**

   Contiene todas las clases que implementan los diferentes algoritmos que se ilustran. Se compone de las siguientes clases:

   a) ServiciosAlgoritmo. Interfaz que contendrá todos los servicios que requerirán los algoritmos para poder mostrar los pasos y el estado del algoritmo. Entre otros métodos, es necesario que la clase que la implemente sea capaz de obtener el peso de las aristas, agregar los pasos del algoritmo, cambiar de paso, marcar los vértices y mostrar la gráfica.

   b) Algoritmo. Es la clase de la que heredan todas las que ilustran algoritmos y contiene varios métodos útiles para mostrar el desarrollo. A continuación se detallan aquellos métodos útiles en la implementación de nuevos algoritmos:

      1) void borraDesarrollo(). Borra la bitácora del algoritmo.

      2) void agregaPasos(String nombre). Agrega los pasos correspondientes al algoritmo actual, para ello es necesario generar imágenes con nombres donde cada una represente un paso del algoritmo que se desea agregar, de tal forma que al ordenarlos alfabéticamente queden ordenados del primer paso al último. Meter todas estas imágenes en una carpeta en

la raíz de donde se encuentren las fuentes. El parámetro que se le pase a este método debe ser el mismo que el de la carpeta que se creo. Es necesario invocar este método en algún momento antes de poder usar el método cambiaPaso, por ejemplo, para el algoritmo BFS, podemos crear una carpeta llamada PasosBFS que contenga los archivos paso1.jpg, paso2.jpg, paso3.jpg, etcétera; en dónde paso1.jpg contenga el primer paso el algoritmo, paso2.jpg el segundo y así sucesivamente. Ejecutar el método agregaPasos con el parámetro "PasosBFS", provocaría que el panel que muestra el algoritmo en pseudocódigo contenga éstas imágenes apiladas en el órden alfabético en el que se encuentran y que el método cambiaPaso sea capaz de cambiar el paso en la imagen paso3.jpg al paso en la imagen paso1.jpg después de ejecutarlo con los parámetros 1 y 3 en ese órden.

3) void agregaADesarrollo(String suceso). Agrega a la bitácora la cadena que se le pasa de parámetro.

4) void cambiaPaso(int a, int b). Cambia en el panel que contiene el algoritmo en pseudo-código, del paso a al paso b; los pasos del algoritmo empiezan en 0, "cambiar" quiere decir que cambia el marco amarillo del paso a al paso b.

5) void marcaArista(int vi, int vj, String color, boolean dirigida). Marca la arista (v_i, v_j) con el color pasado como cadena, que puede ser cualquiera aceptado por Graphviz. Si dirigida es true, será la arista dirigida (v_i, v_j).

6) boolean nulo(int p). Nos dice si el paso p está actualmente marcado.

7) void enviaMensaje(String mensaje). Envía un mensaje en un *popup* al usuario.

8) int numA(int tope). Nos regresa un número aleatorio que va de 0 al entero que se le pasa de parámetro.

9) void dibuja(). Actualiza el dibujo de la gráfica.

10) void marcaVertice(int i, String c). Marca el i-ésimo vértice con el color que se le pase de parámetro, el color puede ir con el modelo de color RGB, por ejemplo el parámetro "0.5,0.8,1" corresponde a un color verde esmeralda. Existen algunos colores que identifica con su nombre en inglés como: red, blue, yellow. [1]

11) void dibuja(int[]d,String[]p). Dibuja la gráfica con distancias y predecesores.

c) AlgoritmoDistancia. Clase de la que heredan los algoritmo BFS y distancias de Dijkstra. Dado que los algoritmos hacen prácticamente lo mismo, esta clase maneja la mayoría de los pasos, excepto por los particulares de cada algoritmo, que son cuando escoge el siguiente vértice a ser usado y, en el caso de distancias de Dijkstra, el paso en donde busca si el peso actual es mejor que el que ya se tiene.

d) BFS. Implementa el algoritmo BFS.

[1] Esto aplica para la versión 2.26 del paquete Graphviz

e) Dijkstra. Implementa el algoritmo de distancias de Dijkstra.

f) CircuitoEuleriano. Implementa el algoritmo de Eüler para encontrar circuitos Eulerianos.

g) DFS. Implementa el algoritmo DFS.

2. **CheckTree**

En java existe una biblioteca llamada JTree que se usa para representar jerárquicamente un conjunto de datos.

El paquete CheckTree modifica el comportamiento de la clase JTree de java añadiéndole en cada nodo un checkbox que es una "caja" que es posible marcar y desmarcar, como se muestra en el manual de diseño de gráficas.

El paquete original fue realizado por Santhosh Kumar y la información adicional puede ser encontrada en la siguiente liga:

`http://www.jroller.com/santhosh/date/20050610`

La clase CheckTreeManager de esta biblioteca fue modificada para cumplir con lo necesario para crear cómodamente gráficas, como se muestra en el manual de diseño de gráficas.

Esta biblioteca se compone de las siguientes clases:

a) CheckTreeCellRenderer. Esta clase controla cómo se muestra cada celda del Jtree

b) CheckTreeManager. Es la encargada de manejar el comportamiento de los JTree de java para que muestren los checkbox y que ocurra el comportamiento mencionado en el manual de diseño de gráficas.

Se le agregaron los siguientes métodos:

1) modificaLista(TreePath t, boolean seleccionado) que recibe de parámetro una TreePath t[2], y un boolean seleccionado. Se encarga de modificar las listas de adyacencias. Si seleccionado es True la Treepath t se agrega a la lista de adyacencias; si no, se quita.

2) agregaTrayectoria(int i, int j, DefaultMutableTreeNode raiz) que marca la TreePath: raiz, i, j que representa a la arista (vi,vj) en los checkboxes correspondientes del árbol.

3) generaGrafica(int v, String c). Regresa una cadena en el lenguaje de Graphviz que corresponde con la información registrada por la interfaz, se usa básicamente para dibujar la gráfica.

[2]Una TreePath es una estructura de datos que representa una trayectoria en un JTree

Hay que mencionar que a esta clase, en su constructor se le pasa de parámetro un JTree y entonces los atributos del parámetro son modificados para que se comporte como se requiere.

c) CheckTreeSelectionModel. Establece el comportamiento de los checkboxes del árbol.

d) TreePathSelectable. Es una interfaz que determina cuando una TreePath es seleccionable.

e) TristateCheckBox. Es una clase que hereda de JCheckBox, controla el comportamiento. de las Checkboxes cuando se les da clic.

3. **Grafica**

Esta biblioteca implementa una gráfica y contiene las siguientes clases:

a) Vertice. Representa a un vértice de la gráfica. Cada vértice tiene su propia **lista de adyacencias** compuesta por vértices. En este caso se prefirió usar una lista pues en algún momento durante la ejecución era necesario eliminar vértices de la lista. Incluye adicionalmente el

b) Grafica. Representa a una gráfica y contiene un arreglo de vértices. Se escogió un arreglo, pues siempre se van a preservar todos los vértices de la gráfica. Esta es la clase que se guarda usando serialización de java. Cuando se carga, de esta clase se genera el JTree correspondiente, permitiendo restaurar un **estado** del programa, la clase JTree no es serializable por eso no se usó en lugar de esta clase.

4. **InterfazPantera**

Este paquete contiene lo necesario para representar gráficamente al programa, tanto la interfaz de java para manejar el programa como una clase que se encarga de manejar lo relacionado con GraphViz. Se compone de las siguientes clases:

a) PanteraView. Es una clase que hereda de FrameView; Es la definición de la interfaz de usuario de este programa y también implementa la interfaz ServiciosAlgoritmo.

b) GraphViz. Esta clase fue tomada de la página de GraphViz, escribe la gráfica a disco en algún formato, en este caso escogimos el formato Gif porque ocupa menos espacio en disco y le agregamos la función de escribir la gráfica en el lenguaje de Graphviz.

Para poder agregar un nuevo algoritmo al programa, se deben cubrir dos aspectos:

1. Añadir una nueva clase que extienda a Algoritmo.
2. Agregar al seleccionador de algoritmos la nueva clase que se generó; agregar al evento que controla el cambio de índice una nueva opción con la clase generada. Por ejemplo, al agregar la clase BFS, se agregó lo siguiente al código de java:

   En el constructor se añadió:

   ```
   seleccionador.add("BFS");
   ```

   y en el método seleccionadorItemStateChanged(java.awt.event.ItemEvent evt), se agregó al switch un nuevo caso

   ```
   case 2:
           alg = new BFS(this);
           break;
   ```

4.2. Graphviz

Como lo mencionan en su página, Graphviz es un software libre para la visualización de gráficas. Toma descripciones cortas de gráficas escritas en un archivo de texto plano. Es capaz de realizar diagramas en muchos formatos útiles, como imágenes y SVG para páginas web, Postscript para ser agregado en PDF o quizá otros documentos.

Graphviz tiene muchas características interesantes para realizar gráficas, como lo son: opciones de color para vértices y aristas; fuentes de texto; estilos de línea; formas personalizadas para los nodos; capacidad de agregar texto a los nodos, entre otras.

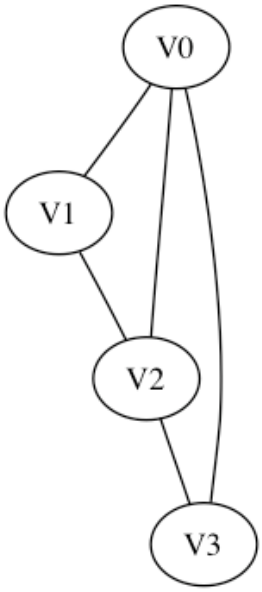
Además el lenguaje es bastante intuitivo y fácil de usar, como podemos ver en el código en la figura 4.1:

Figura 4.1: Código de ejemplo en el lenguaje de graphviz

```
digraph G {
V0[shape=ellipse, width=.5, height=.5];
V1[shape=ellipse, width=.5, height=.5];
V2[shape=ellipse, width=.5, height=.5];
V3[shape=ellipse, width=.5, height=.5];
V0->V1[arrowhead=none];
V0->V2[arrowhead=none];
V0->V3[arrowhead=none];
V1->V2[arrowhead=none];
V2->V3[arrowhead=none];
center = true;
}
```

Genera la siguiente gráfica:

Figura 4.2: Gráfica generada por el código de la figura 4.1



Capítulo 5

Conclusiones

5.1. Experiencia haciendo el proyecto

Al comenzar el proyecto, me sorprendió que no existan paquetes gratuitos multiplataforma en la actualidad que realicen la funcionalidad conseguida. Buscando en internet, sólo encontré applets de java que presentan uno o más de los siguientes problemas:

1. Ejecutan ejemplos pero para gráficas definidas de antemano.
2. No arrojan suficiente información acerca de lo que está ocurriendo durante el proceso.
3. Están escritos en inglés.
4. La interfaz para el diseño de la gráfica no es suficientemente buena.
5. Ilustran sólo un algoritmo.

Algunos ejemplos de applets relacionados son:

- Ejemplo de BFS:
  `http://www.cs.usask.ca/content/resources/csconcepts/1998_3/BFS/java/index.html`
- Ejemplo de DFS:
  `http://www.cs.usask.ca/content/resources/csconcepts/1998_3/DFS/java/index.html`
- Ejemplo de Dijkstra:
  `http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html`

Creo que la razón por la que no existen programas parecidos con la madurez suficiente reside en las personas que son candidatos a quererlo hacer: profesores o ayudantes de clase que son personas que no tienen mucho espacio en su agenda para programar algo que requiere tiempo y esfuerzo, pero que considero es una herramienta importante para la enseñanza.

El desarrollo del programa fue agradable para mí, pues adquirí experiencia en el uso de componentes de java para el desarrollo de interfaces, el manejo de eventos, el uso de componentes contenidos en otros componentes, inserción de componentes durante la ejecución del programa, manejo de imágenes, etcétera.

5.2. Beneficio de este tipo de ayudas

Al entrar a la Universidad, la gran mayoría de los alumnos, vienen con muy poca madurez matemática, complicándoles el entendimiento de los diferentes temas que se ven a lo largo de los primeros semestres.

Para solucionar este problema se han realizado numerosos esfuerzos, desde libros con muchas ilustraciones hasta el uso de diferentes materiales didácticos.

Pero programas que ayuden al estudiante a entender mejor el funcionamiento de algoritmos en este nivel básico no se habían realizado antes, por eso decidimos realizar el sistema.

El desarrollo del paquete comenzó con la idea de hacer algo útil para los alumnos de cursos introductorios que revisen algoritmos de teoría de gráficas, para que puedan entender mejor el funcionamiento de lo ilustrado en los teoremas, pero también para que aprendan a seguir un algoritmo que son dos cosas fundamentales y que usarán los alumnos de ciencias de la computación durante toda la carrera.

Por los que se están preguntando, al sistema decidí nombrarlo **Pantera** porque la piel de dicho animal asemeja un *manto de estrellas*. Las estrellas del cielo pueden ser representadas por vértices y ciertas agrupaciones de estrellas forman constelaciones que para poder entender mejor el nombre que se les asigna, es necesario unir correctamente pares de estrellas que pueden representarse como aristas.

Bibliografía

[1] David Gries y Fred B.Schneider. *A Logical Approach to Discrete Mathematics.* Springer-Verlag, 1994.

[2] Documentación de java: [1]

`http://java.sun.com/j2se/javadoc/`

[3] . J.A. Bondy and U.S.R. Murty. *Graph Theory with Applications.*

[4] . Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein. *Introduction to algorithms.* Mc Graw Hill, 2004.

[5] Gary Chartrand. *Introduction to Graph Theory.* Dover Books on Advanced Mathematics.

[6] Favio Miranda y Elisa Viso. *Matemáticas Discretas.* Las Prensas de Ciencias, 2009. Página del libro: `http://lambda.fciencias.unam.mx/~elisa/NotasDeCursos/Discretas/`

[7] Página de Graphviz: `http://www.graphviz.org/`

[8] Página de referencia para la clase original de CheckTree: `http://www.jroller.com/santhosh/date/20050610`

[1] Fecha de consulta de las páginas web: 11 de agosto de 2010