UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIAS MATEMÁTICAS

FACULTAD DE CIENCIAS

GRÁFICAS Y PRUEBAS

T E S I S

QUE PARA OPTAR POR EL GRADO ACADÉMICO DE

MAESTRO EN CIENCIAS (MATEMÁTICAS)

P R E S E N T A

JORGE ALCALDE MARTÍN DEL CAMPO

DIRECTORA DE TESIS: DRA. HORTENSIA GALEANA SÁNCHEZ
INSTITUTO DE MATEMÁTICAS

México, Cd. Mx. **MARZO, 2018**

Gráficas y Pruebas

Jorge Alcalde Martín del Campo

para Ella y ella sabe por qué

Índice

If you're not having fun, you're doing something wrong.
Groucho Marx

Agradecimientos

En estos casos siempre deja uno a alguien fuera, son ustedes los primeros a los que les agradezco el apoyo para la realización de este trabajo.

Gracias Hortensia por confiar, apoyar y mejorar este proyecto.

Gracias a todo el equipo: Juancho, Mika, Ingrid y Guadalupe, les agradezco todo su tiempo, comentarios y el apoyo para realizar este trabajo y sobre todo a mi.

Agradezco a la UNAM, a la Facultad de Ciencias y a todo el personal de posgrado, por toda la paciencia, apoyo y permitirme, al fin, terminar con este ciclo.

Gracias madre por seguir apoyando a tu hijo.

Gracias en especial a Víctor Neumann quien me enseñó a amar el arte de la Teoría de Gráficas.

Contigo siempre se cubren todas mis trayectorias. Como siempre y por siempre, gracias Ellita ...

Beauty is the first test: there is no permanent place in the world for ugly mathematics.
G. H. Hardy

Introducción

Todo necesita probarse. Las primeras acciones que hace un bebé son las de tocar, probar y comprobar todo lo que encuentra, es la manera natural de aprender a interactuar con el mundo que nos rodea. Desafortunadamente, al ir creciendo vamos perdiendo la curiosidad y necesidad de comprobarlo todo. Es posible que sea un problema social, debido a que al ir creciendo, aumenta también nuestro miedo a las pruebas, es decir, si vamos a una cita con el doctor (¡me voy a morir!), o si hacemos un cuestionario sicológico (¡estoy loco!), o una muestra de antidoping, embarazo, o si vemos alguna comida que a la vista no sea agradable, nos da miedo y no comprobamos si realmente es lo que creíamos.

Sin embargo, en todas las actividades a las que nos enfrentamos, por ejemplo al proponer un teorema matemático, tenemos que comprobarlo; si compramos un coche tenemos que verificar que funcione correctamente (o al menos asumir que alguien ya lo hizo por nosotros). En fin, para que las cosas funcionen como debe ser, hagan lo que tengan que hacer, o podamos creer en lo que enuncian, debemos de comprobarlo.

Los sistemas de cómputo deben ser probados y probados bien, ya que no podemos depender solamente de la habilidad del programador que lo desarrolla. El programa o función para lo que fue construido debe de trabajar adecuadamente. Con frecuencia confiamos ciegamente en los sistemas de cómputo, como por ejemplo los sistemas de control de tráfico aéreo o los programas que vienen incluidos en aparatos tales como teléfonos celulares o reproductores de música. Sin embargo, para que hagan lo que tienen que hacer, alguien tuvo que probarlos primero. Hay otros ejemplos, como los sistemas bancarios que, aunque funcionen, se prueben y los usemos cotidianamente, no necesariamente les tenemos confianza, pero eso es otra historia.

En este trabajo hablaré sobre algunas de las pruebas de software y algunos resultados matemáticos relacionados con la Teoría de Gráficas. Aunque existe una gran diversidad de pruebas, el revisarlas todas esta fuera del objetivo y alcance del presente análisis.

Así, la pregunta principal de este trabajo es:

¿Dónde encontramos Teoría de Gráficas en las pruebas de software?

Antecedentes

Abordaré dos caminos que se juntarán y separarán por momentos, la parte de Teoría de Gráficas y la parte de pruebas de software. Trataré, donde me sea posible, de no profundizar demasiado con términos técnicos en ninguno de ellos, ya que mi intención es enseñar solamente que existe Teoría de Gráficas escondida o inherente a las pruebas de software.

Curiosamente, tanto el desarrollo de la Teoría de Gráficas como el de pruebas de software son ramas "jóvenes" dentro de sus respectivos campos: en particular, el desarrollo formal de la Teoría de Gráficas es posterior y consecuencia de varios problemas o juegos matemáticos que permitieron generar muchos resultados interesantes, algunos de los cuales tocaremos brevemente.

Cuando me refiero a Teoría de Gráficas, el término gráfica no debe de confundirse con “gráfica” en el sentido de gráfica de pay, gráfica de barras, o gráfica de esfuerzo. En el caso que nos ocupa, se refiere a un término matemático que más adelante definiré formalmente para dejar claro este punto.

La Teoría de Gráficas es una rama de las matemáticas que actualmente tiene aplicaciones, entre otras, en química, genética, ingeniería, administración, sociología y ciencias de la computación. Una aplicación muy conocida que usa Teoría de Gráficas es MapQuest, un sitio popular para encontrar información, como direcciones de manejo o las trayectorias mas rápidas para llegar de un punto a otro.

La Teoría de Gráficas puede decirse que empieza en 1736 en el pueblo Prusiano de Königsberg (actualmente Kaliningrado, Rusia). El pueblo está sobre las orillas del río Pregel y tiene dos islas en medio del río. Los domingos por la tarde los ciudadanos de Königsberg acostumbraban pasear a través de los siete puentes que unían las diferentes partes del pueblo y las islas.

Un pasatiempo común para los habitantes de Königsberg era tratar de pasar por los puentes exactamente una sola vez y regresar al punto de partida. En la figura 1 podemos ver cómo estaban distribuidos los puentes.

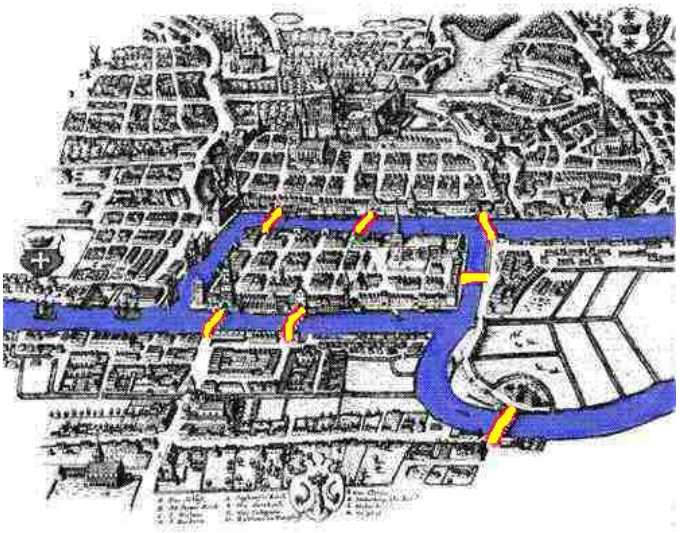


Figura 1. Puentes de Königsberg (tomada de Internet)

Después de varios años de no lograr el objetivo de no pasar por los puentes más de una vez, le preguntaron a un matemático llamado Leonard Euler si ello era posible.

En una traducción libre, y simplificada para este texto, del trabajo de Euler *"SOLUTIO PROBLEMATIS AD GEOMETRIAM SITUS PERTINENTIS"*, tomado de la traducción de Biggs, Lloyd y Wilson (1976), Euler comienza buscando la solución de la siguiente manera:

"El problema, como me fue contado es conocido de la siguiente manera, en Königsberg hay una isla A, llamada Kneiphof; el río que la rodea está dividido en dos brazos y estos brazos están cruzados por siete puentes, a, b, c, d, e, f y g. Con respecto a los puentes, se busca ver si es posible que alguien encuentre una ruta de tal manera que cruce los puentes una y solamente una vez. Se me ha dicho que hay personas que dicen que esto es imposible, mientras otras dudan de ello, pero nadie garantiza que pueda o no hacerse. De este hecho he formulado el problema general: Cualquiera que sea el arreglo y la división del rió en brazos y no importando cuántos puentes existan, ¿se puede encontrar cuándo es posible o no, cruzar los puentes exactamente una vez?"

Euler comienza el análisis del problema reemplazando el mapa de la ciudad con un diagrama más simple, resaltando la característica principal. En Teoría de Gráficas lo simplificamos más todavía para incluir solamente puntos (que representan masas de tierra) y líneas (representando puentes)

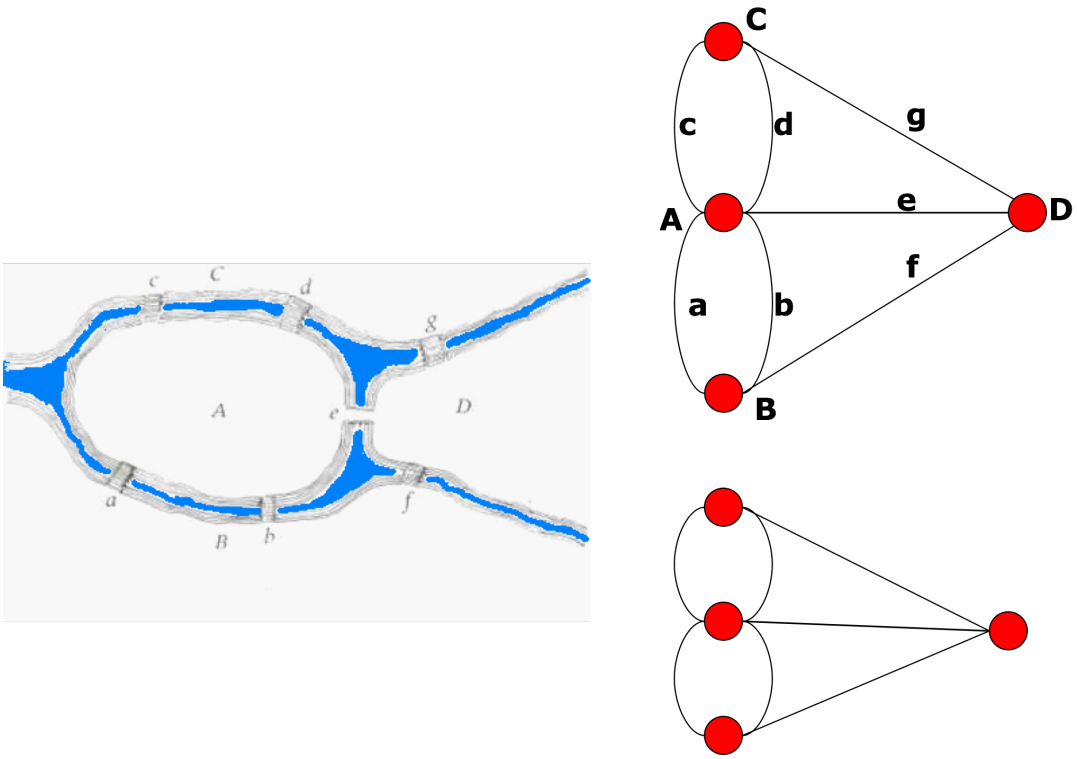


Figura 2. Modelo de Euler y gráficas que representan el problema de los puentes de Königsberg

Regresando al trabajo de Euler en párrafos siguientes comenta:

"El problema de los puentes puede ser resuelto haciendo una lista exhaustiva de todas las rutas posibles, y de ahí ver si una ruta cumple con las condiciones del problema, debido al número de posibilidades, este método es difícil y laborioso y en otros problemas con más puentes podría ser imposible, por lo que me concentraré sólo en el problema de si la ruta puede, o no, ser encontrada. Mi método se basa en la manera de representar el cruce por un puente, para esto uso letras mayúsculas para las áreas de tierra separadas por el río, si un viajero va de A a B por el puente a o b, simplemente lo escribo AB, donde la primer letra es donde sale el viajero y la segunda donde llega, en general, no importando cuantos puentes cruce un viajero, su jornada es representada por un número de letras mayúsculas mayor en uno que el número de puentes, por lo que el cruce de siete puentes requiere ocho letras para representarlo."

No traduciré todo el texto, ya que no es el objetivo de este documento, lo más importante es ver cómo Euler modela el problema y logra encontrar un resultado general no importando el número de puentes. El punto central para llegar al resultado se encuentra en las áreas, donde Euler verifica que para entrar y salir de un punto, se requiere un número par de "visitas". Euler termina diciendo:

“Cualquiera que sea el problema propuesto, uno puede determinar fácilmente si un viaje cruzando un puente una sola vez es posible, con las siguientes reglas:

Si hay más de dos áreas en las cuales un número impar de puentes lleguen, entonces el trayecto es imposible.

Si el número de puentes es impar, en exactamente dos áreas, entonces el viaje es posible si empieza en una de esas áreas y terminas en la otra.

Finalmente, si no hay áreas con un número impar de puentes, entonces es posible hacer el trayecto empezando en cualquier punto y terminando en el mismo.”

En el caso de los puentes de Königsberg, cada uno de los vértices (áreas) tiene un número impar de uniones (puentes), por lo que es imposible que un caminante pueda completar el recorrido sin cruzar los puentes más de una vez.

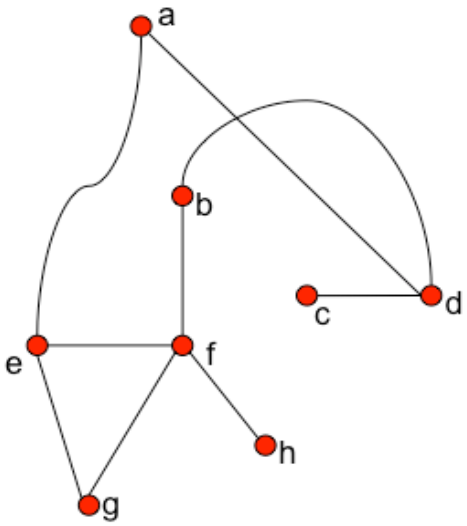
Antes de continuar con la historia y adentrarnos al tema de interés, daré unas definiciones formales e informales y algunos teoremas sobre la Teoría de Gráficas, que servirán para entender mejor estos conceptos y nos ayudarán en nuestras tareas de pruebas.

Gráficas

Una gráfica *G* es un par ordenado de conjuntos (*V(G), A(G)*), donde *V(G)* es un conjunto finito de objetos llamados vértices (o nodos) y los elementos de *A(G)* son un subconjunto de pares no ordenados de *V(G),* llamados aristas (o líneas o arcos). Cuando no se preste a confusión *V(G)* será simplemente *V* y *A(G)* será *A*.

Se llaman gráficas dado que tienen una propiedad importante, que podemos dibujarlas, hay que notar que no hay una manera única de hacerlo, la posición relativa de los vértices y la línea entre las aristas no tiene importancia. La manera usual de hacerlo, es pintando un punto o un círculo por cada vértice, uniendo dos de estos puntos con una línea si dicho par forma una arista, como se ve en la figura 3.

V(G) = {a, b, c, d, e, f, g, h}
A(G) = {(a,e), (a,d), (b,d), (c,d), (b,f), (e,f), (e,g), (f,g), (f,h)}



V(G) = {a, b, c, d, e, f, g}
A(G)= {(a,b), (a,c), (c,d), (c,e), (c,f), (f,g)}

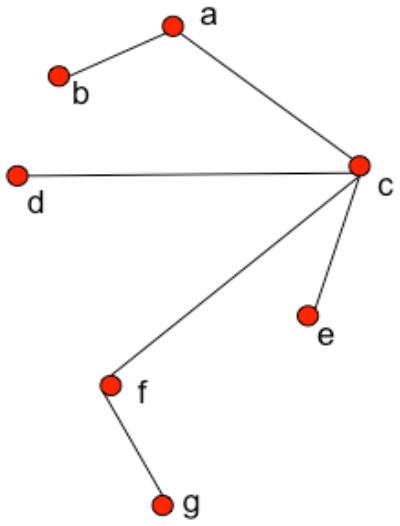


Figura 3. Gráficas, donde los puntos representan a los vértices y las líneas a las aristas.

Otro punto a considerar en una gráfica dibujada es que dos aristas pueden cruzarse en un punto que no esté en la gráfica, como por ejemplo las aristas (b,d) y (a,d) de la figura 3. Aquellas gráficas que podamos dibujar en las que ninguna arista corte a otra, salvo en los vértices, es llamada una gráfica plana.

Algunos ejemplos de representaciones por medio de gráficas pueden ser las redes de transporte, donde los vértices de la gráfica serían ciudades y las aristas las rutas de transporte entre ellas, ya sea de aviones, camiones, carreteras, etcétera. Otro ejemplo de representación puede ser el de redes de información como la web, donde los vértices son las páginas web y las aristas se definen si existe una liga (o hiper-liga) entre éstas; o las redes sociales donde los vértices son personas y las aristas relaciones entre ellas. Las gráficas nos pueden ayudar a representar gran diversidad de actividades tanto técnicas como humanas.

En una gráfica, cada arista puede además tener un valor (a veces llamado peso) asociada a ella; en ese caso decimos que tenemos una gráfica con peso como en la figura 4a.

Una arista a = (u,u) se llama bucle o rizo, una gráfica con bucles o rizos es llamada pseudográfica (figura 4c).
Una arista que aparece repetida en A se llama arista múltiple, una gráfica con aristas múltiples se llama multigráfica (figura 4b).

Una gráfica sin bucles, ni aristas múltiples, se llama gráfica simple.

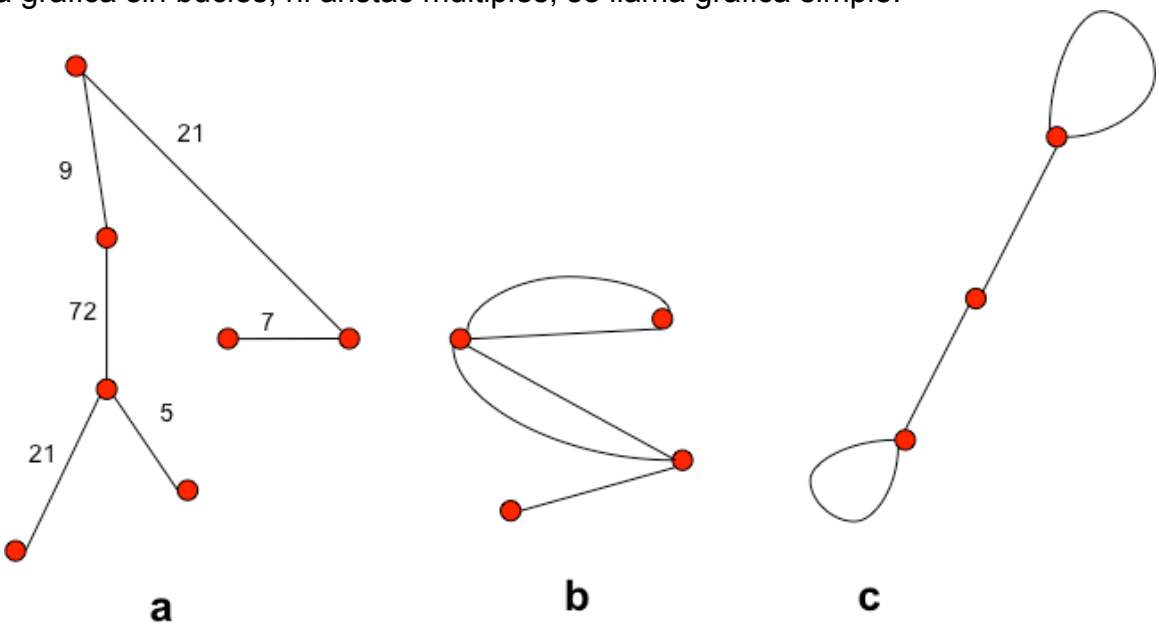


Figura 4. Gráficas: con peso (4a), multigráfica (4b) y pseudográfica (4c).

Una Digráfica *D* es un par de conjuntos (*V(D), F(D)*), donde *V(D)* es un conjunto finito de objetos llamados vértices y los elementos de *F(D)* son un subconjunto de pares ordenados de *V(D),* llamados flechas (ver figura 5a).

Una gráfica es completa si todo vértice está conectado con otro, y se le denomina K_n, donde n es el número de vértices (ver ejemplo de la figura 5b). La K es en honor al matemático polaco Kazimierz Kuratowski, pionero en el estudio de la Teoría de Gráficas.

V(D) = {1, 2, 3, 4, 5, 6, 7}
F(D) = {(1,2), (3,1), (2,4), (4,7), (6,7), (6,5), (5,7)}

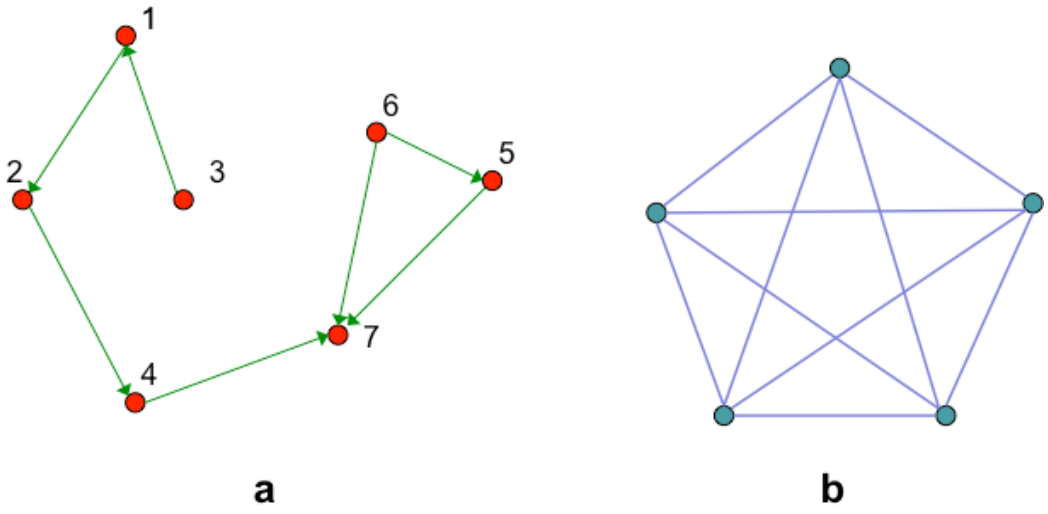


Figura 5. Digráfica (5a) y gráfica completa de 5 vértices K_5 (5b)

Dos vértices v, w son adyacentes (o vecinos) si la arista vw está en la gráfica. El grado (o valencia) de un vértice u, es el número de aristas que inciden en dicho vértice, o basado en la definición anterior, es el número de vecinos que tiene y lo denominaremos gr(u).

Un camino C es una sucesión de vértices, digamos que $C = (u_0, u_1, ..., u_n)$, tal que u_i es adyacente a u_{i+1} para cada $0 \leq i \leq n-1$. La longitud del camino C es n, es decir, el número de aristas que recorre dicho camino, el cual vemos que puede pasar varias veces por la misma arista. Así, lo que cuenta es el número de veces que un camino pasa por éstas, no el número de aristas. El camino C es cerrado cuando $u_0 = u_n$. Notemos que un camino puede repetir aristas y vértices, es decir, pasar varias veces por las mismas aristas y los mismos vértices.

Una trayectoria es un camino en el que no se repiten vértices (figura 6a). La distancia entre dos vértices es la mínima de las longitudes de las trayectorias entre ellos, el diámetro de una gráfica es la mayor distancia de entre todos los pares de vértices de la misma.

Un ciclo es un camino de longitud al menos tres, en el cual no se repiten vértices excepto el primero y el último que son iguales (figura 6b).

Un paseo es un camino el que no se repiten aristas (pero podrían repetirse vértices).

Un paseo Euleriano, es un paseo que pasa por todas las aristas de la gráfica. Una gráfica es Euleriana si admite un paseo Euleriano cerrado.

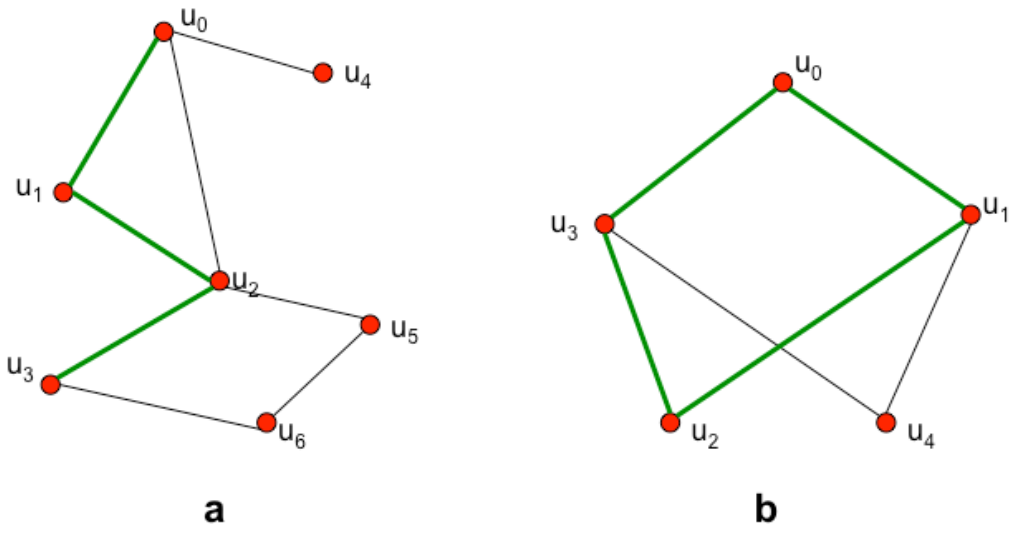


Figura 6. Una trayectoria (u_0, u_1, u_2, u_3) (6a) y un ciclo (u_0, u_1, u_2, u_3, u_0) (6b)

Una gráfica G es conexa si para cualquier par de vértices u,v de G existe una trayectoria entre ellos. Notemos que en esta definición podemos substituir trayectoria por camino. Toda gráfica con un solo vértice es conexa.

Dada una digráfica *D*, definimos una trayectoria *T* en *D* como una sucesión de vértices $T = (u=u_0, u_1, ..., u_n=v)$ donde $u_i \neq u_j$ para todos $i \neq j$ y $(u_i, u_{i+1}) \in F(D)$ o $(u_{i+1}, u_i) \in F(D)$, la trayectoria T en D es una trayectoria dirigida, siempre que $(u_i,u_{i+1}) \in F(D)$, y le llamamos una uv-trayectoria dirigida. Recordemos que en una digráfica, las flechas al ser pares ordenados tienen una dirección, es decir, la flecha (u,v) es distinta a la flecha (v,u), por lo que podría haber una uv-trayectoria pero no necesariamente una vu-trayectoria.

Una digráfica D es fuertemente conexa si para cualesquiera $u, v \in V(D)$ existe una uv-trayectoria dirigida y una vu-trayectoria dirigida.

Si D es una digráfica y v ∈ V(D), el exgrado de v, denotado por $\delta^+D(v)$, es el número de vértices adyacentes desde v (flechas de salida) y el ingrado de v, denotado por $\delta^- D(v)$, es el número de vértices adyacentes hacia v (flechas de entrada).

En todos los ejemplos de este trabajo se asume que las gráficas son conexas o coloquialmente, "de una sola pieza".

Una gráfica conexa que no tiene ciclos se llama árbol (figura 7). Una característica muy importante de los árboles es que el número de aristas de cada árbol es igual al número de vértices menos 1, es decir, todo árbol de n vértices tiene $n - 1$ aristas.

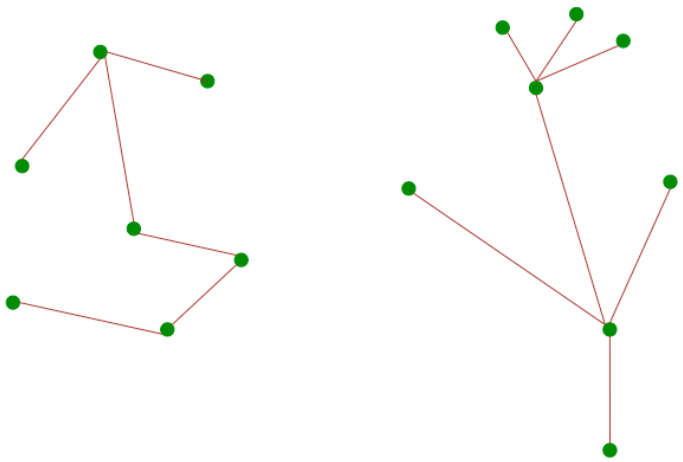


Figura 7. Gráficas conexas sin ciclos llamadas árboles

Sabemos que un árbol no tiene ciclos, pero si a un árbol le adicionamos una arista cualquiera, creamos exactamente un ciclo.

Sea G una gráfica conexa cualquiera y definamos el número ciclomático de G (mu) como el menor número de aristas que se le tienen que quitar a la gráfica de e aristas y n vértices para que no quede ningún ciclo y obtengamos un árbol; la fórmula para obtener este número está dada por:

$mu = e - n + 1$

Una parte sorprendente de la Teoría de Gráficas es la cantidad de resultados diferentes y complejos que se obtienen únicamente con estas definiciones. En particular podemos probar un primer resultado, el cual relaciona el número de aristas (tamaño) en una gráfica y los grados de sus vértices.

Si sumamos los grados de todos los vértices de una gráfica se puede ver que cada arista e = uv se cuenta dos veces, una vez cuando contamos el grado de u y otra cuando contamos el grado de v.

De aquí podemos formular el siguiente teorema:

Teorema. En una gráfica G, la suma de los grados de sus vértices es el doble del número de aristas. Es decir, si G = (V, A) es una gráfica, entonces

$$\sum_{u \in V} gr(u) = 2|A|$$

El teorema anterior también es conocido como el Teorema o Lema de los Apretones de Manos (Handshaking Lemma), pues se puede formular de la siguiente forma:

“En cualquier reunión, el número total de manos que se aprietan cuando las personas se saludan, es par”. Podemos pensar que en cualquier gráfica un “apretón” es una arista y cada persona (mano) es un vértice, por lo cual cada apretón cuenta dos veces, una por cada persona que los da o recibe.

Aprovechando este teorema, podemos probar el siguiente corolario:

Corolario. En una gráfica el número de vértices de grado impar es par.

Demostración. Sea G una gráfica con e aristas. Por el Teorema anterior sabemos que:

$$\sum_{u \in V} gr(u) = 2e$$

Separando la suma en dos términos, correspondientes a los vértices de grado par y los vértices de grado impar, tenemos:

$$\sum_{u \in V \ gr(u) \text{ par}} gr(u) \quad + \quad \sum_{u \in V \ gr(u) \text{ impar}} gr(u) \quad = 2e$$

Como el primer sumando sólo esta formado por la suma de números pares, se sigue que el resultado es un número par. Sabemos por el Teorema que la suma total también es par, por lo tanto, tenemos que el segundo sumando también tiene que ser par, y esto sucede si y sólo si el número de vértices de grado impar es par ■

Aquí termino con las definiciones básicas y necesarias; daré unas cuantas definiciones más a lo largo del texto, pero las principales son las enlistadas arriba. Antes de proseguir hacia la parte de pruebas, regresaré un poco a la historia de la Teoría de Gráficas.

Euler (1707-1783) escribió y publicó más de 900 artículos, de los cuales, más de la mitad los hizo estando completamente ciego. Sorprendentemente, a pesar de ser considerado uno de los más grandes matemáticos de la historia, nunca publicó un algoritmo para encontrar un paseo Euleriano en una gráfica. Sin embargo, desarrolló herramientas para determinar si dicho paseo existía o no. Un algoritmo para encontrar un paseo Euleriano está descrito por el matemático francés Fleury desde 1883, por lo que no entraré en el detalle de cómo encontrarlo, sino que asumiré que el paseo existe y que podemos obtenerlo.

En 1962, 226 años después de Königsberg, un matemático chino llamado Kwan Mei-Ko se planteó la siguiente pregunta: dado que es imposible cruzar todos los puentes exactamente una vez y regresar al punto inicial ¿cuál será el mínimo de recruces necesario?

Este es el tipo de problema al que un cartero se enfrenta cotidianamente cuando entrega la correspondencia. Supongamos que cada una de las aristas de la gráfica de la figura 8 es una calle donde un cartero debe de entregar correo, y cada uno de los vértices es una intersección de calles donde el cartero puede cambiar de dirección.

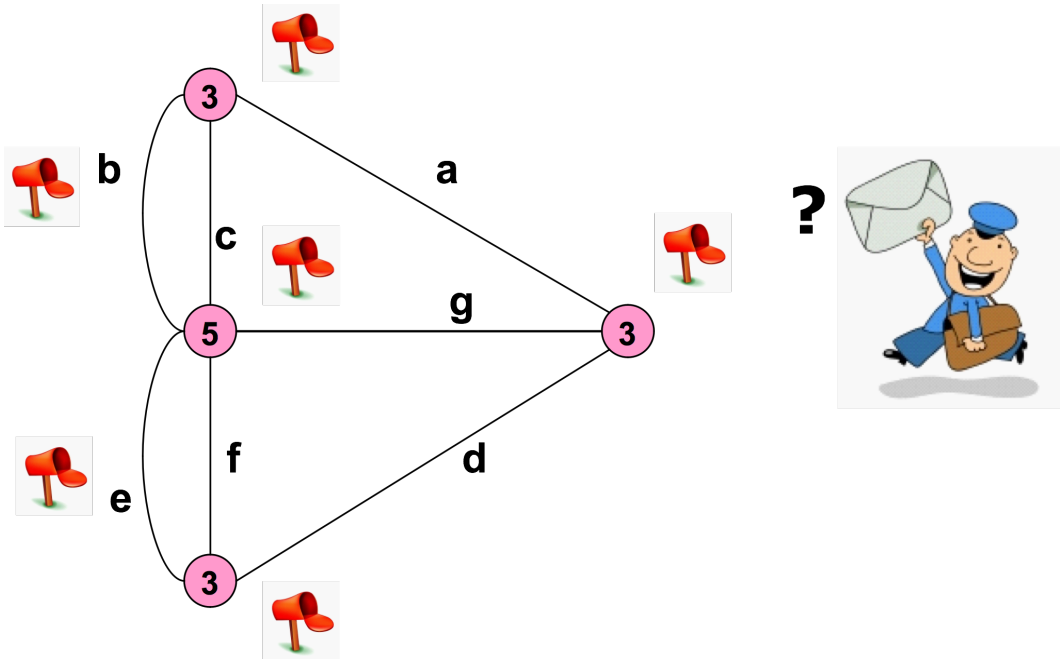


Figura 8. El problema del cartero chino

Para ayudar a entender este problema, en la figura 8 añadí a cada vértice el grado que le corresponde (número de vecinos).

Recordemos que en una gráfica, el número de vértices de grado impar es par, en este caso hay cuatro vértices de grado impar, lo que buscamos, según el resultado de Euler es tener todos los grados pares, podemos duplicar todas las aristas, con lo que automáticamente los grados de todos los vértices serían pares y, así podríamos

encontrar un paseo que pasara por todos los puentes. Sin embargo, como se busca el mínimo número de recruces necesario, la solución de Kwan fue usar el Corolario anterior y adicionó una arista (multiarista) por cada recruce de grado impar de la ruta en la gráfica. En este ejemplo, hay cuatro vértices de grado impar, por lo que con solamente aumentar dos aristas se pueden poner todos los vértices de grado par.

Estas dos aristas "duplicadas" o virtuales, como se ve en la gráfica de la figura 9, modifican a cada vértice con grado par; a este proceso le llamaremos "Eulerización".

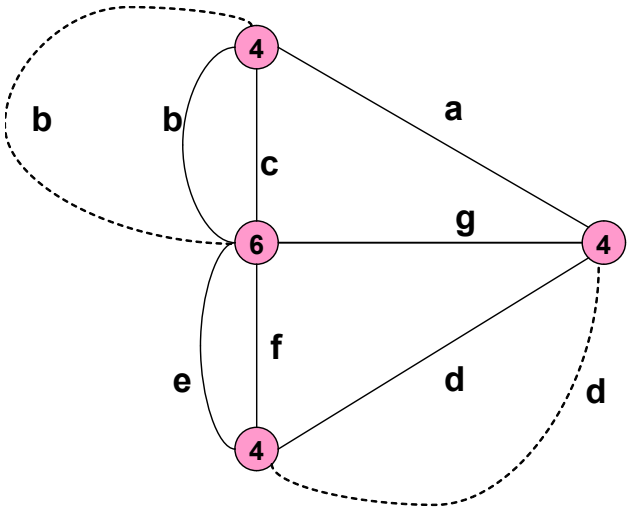


Figura 9. Solución de Kwan al problema del cartero.

Una vez que la gráfica es "Eulerizada" se puede obtener un paseo Euleriano. En el caso de nuestro cartero, él podría tomar esta ruta mínima para entregar el correo:

a b c b e f g d d

repitiendo en el trayecto de su ruta solamente los segmentos (calles en el caso del cartero o puentes en el problema original) *b* y *d*.

Este problema es conocido en honor de Kwan como el problema del cartero chino, y nos sirve para entender lo que hay detrás de ciertas técnicas de pruebas.

Pruebas de software

Es necesario siempre realizar pruebas para confirmar el buen funcionamiento de un sistema, ya que los sistemas son hechos por humanos y los humanos tendemos a cometer errores. Una falla en un sistema que ya está liberado o en producción es muy costosa, por lo que se busca detectar los defectos desde un inicio.

La empresa austriaca de pruebas de software, Tricentis, en su reporte del 2016 estima que el costo acumulado el año pasado de errores de software, problemas técnicos y fallas de seguridad en todo el mundo fue de la friolera de $1,1 billones de dólares. Desde cierta perspectiva, eso es más que el producto interno bruto (PIB) total de México el año pasado. Los investigadores revisaron 548 fallas de software que encontraron en artículos de noticias en inglés. En total, las fallas en unas 363 empresas afectaron a unos 4,400 millones de clientes y causaron más de 315.5 años de pérdida de tiempo, según ese informe.

La revista Computerworld en su edición de junio de 2002, en un artículo de Patrick Thibodeau, estimaba que solamente en Estados Unidos los errores de software costaban alrededor de 60 mil millones de dólares al año. El autor comenta además que mejoras en las pruebas podrían reducir el gasto en una tercera parte; sin embargo reconoce que nunca se podrán eliminar por completo los errores y, por ende, los costos.

Las pruebas miden la calidad del software (o como hubiera dicho un olvidable político mexicano, ni la aumentan ni la disminuyen, sino todo lo contrario). Asimismo, las pruebas validan la funcionalidad del software, además de atributos no funcionales como son la confiabilidad, facilidad de uso y mantenimiento.

En un enfoque tradicional de pruebas, la meta generalmente es mostrar que el sistema trabaja adecuadamente, para lo cual se usan casos de prueba simples. Así, se consideran exitosos si el sistema trabaja, sin embargo, este enfoque comúnmente deja muchos defectos escondidos. En un enfoque mejorado, la meta debería ser encontrar defectos con casos de prueba diseñados sistemáticamente para ello, con lo cual el éxito sería encontrar dichos defectos o confirmar que el software está libre de ellos.

No pretendo en este trabajo hacer un compendio sobre pruebas de software ya que existe mucha literatura al respecto; recomiendo sobre todo el libro de Koomen y Pol (1999) como una guía paso a paso para el proceso de pruebas estructurado. Además, incluyo en el apéndice C una breve línea de tiempo de algunos sucesos importantes en el proceso de pruebas de software.

Existen muchas metodologías y pruebas de las que no hablaremos, solamente me referiré a algunas pruebas de software y su relación con la Teoría de Gráficas.

Errores de software

A continuación describo solamente algunos ejemplos de fallas de software que no incluyen virus, sino simples errores en el software exponiendo un error común de percepción, donde los optimistas piensan que una vez que el software funciona bien, funcionará bien siempre.

En febrero de 2017, una falla catastrófica que afectó a cinco hospitales australianos del estado de Queensland fue introducida durante la aplicación de parches de seguridad diseñados para contrarrestar futuros ciberataques. Se requirió más de dos semanas para que los hospitales recuperaran sus sistemas de registros médicos electrónicos, ya que fueron borrados del sistema con esta "mejora".

El 1 de agosto de 2012, Knight Capital causó una importante interrupción del mercado de valores que generó una gran pérdida para la compañía. El incidente ocurrió debido a que un técnico olvidó copiar el nuevo código del Programa de liquidez al por menor (Retail Liquidity Program, RLP) a uno de los ocho servidores de su computadora SMARS, que era el sistema de enrutamiento automático que usaba Knight para pedidos de acciones. El código RLP reutilizó una bandera que anteriormente se usaba para activar la antigua función conocida como 'Power Peg'. Power Peg está diseñado para subir y bajar los precios de las acciones con el fin de verificar el comportamiento de los algoritmos de negociación en un entorno controlado. Debido a esto, los pedidos enviados con la bandera reutilizada al octavo servidor, desencadenaron el código defectuoso Power Peg que todavía estaba presente en ese servidor. Ese mismo día, las acciones de la compañía cayeron un 33 por ciento; al día siguiente, el 75 por ciento del valor de equidad de Knight había sido borrado.

Una serie de tragedias causadas por fallas de software demuestran que aunque un software esté funcionando, puede seguir teniendo errores, como los ejemplos que detallo a continuación (las primeras las tomé de Jaintao Pan, 1999, y la última de Linda Geppert, noviembre 2004, publicada en la revista digital Spectrum). Existen muchos más ejemplos y se puede escribir un libro completo sobre fallas de software, pero con estos ejemplos pienso que la necesidad de hacer pruebas queda más que claro.

El software puede tomar decisiones, las cuales pueden ser tan inseguras como los seres humanos: el 5 de mayo de 1982 en la llamada guerra de las Malvinas, el destructor inglés Sheffíeld fue hundido porque su sistema de radar identificó a un misil enemigo como '"amistoso".

Una máquina de terapia de radiación llamada Therac-25, producida por la compañía AECL (Atomic Energy of Canadá Limited), estuvo involucrada en al menos seis accidentes entre 1985 y 1987, tres de ellos fatales, en los cuales los pacientes recibieron dosis masivas de radiación cien veces mayores a las recetadas. Las dosis fueron causadas por un error de software que no detectó una condición específica. A partir de entonces y gracias a la alarma que provocó dicho evento en el departamento de salud de Canadá, se creó un nuevo estándar de seguridad informática. Sin embargo,

acciones como ésta nos alertan que es peligroso abandonar la vieja y tradicional escuela de seguridad manual, y aún más peligroso depender completamente de mecanismos controlados por software.

El 25 de febrero de 1991, durante la Guerra del Golfo, un misil Patriot falló al intentar detener un misil iraquí Scud, causando la muerte de 28 soldados. Un reporte oficial del suceso lo atribuye a una falla del software de control, y que la causa fue un cálculo incorrecto del tiempo de inicio, debido a errores en cálculos aritméticos de tiempo. El reloj interno del sistema "cortaba" cadenas de tiempo que podrían parecer no importantes de alrededor de 0.000000095. Al tener encendido el sistema por más de 100 horas, el error de corte ya era de 0.34 de segundo, y hay que tomar en cuenta que un misil Scud recorre más de medio kilómetro en ese lapso.

Un corolario particular a este error de percepción es el de pensar que si se prueba bien y el software no falla, no fallará más, o que si falló y se corrigió dicho error y se pone a trabajar de nuevo, ya no fallará.

En el verano de 1991 la compañía DSC Communications de Plano, Texas, detectó un error en el software telefónico desarrollado por ellos e instalado por todos los Estados Unidos. Decidieron corregirlo mandando una actualización del software, ya que tendría muchas mejoras, pero se equivocaron, pues la nueva versión del software deshabilitaba la capacidad de manejar tráfico pesado en la red telefónica. Tardaron semanas en encontrar el nuevo problema, causando un caos de comunicaciones en el sur de California, Pittsburg y Washington D.C.

El 4 de junio de 1996 el cohete espacial Ariane 5 despegó de la base europea en la Guyana Francesa y explotó cuarenta segundos después. El costo del proyecto más el valor del cohete fue mayor a los siete mil quinientos millones de dólares. Al publicarse el reporte de las causas de la explosión, se detectó un error en el software del sistema de referencia interna. Un número relativo a la velocidad horizontal con respecto a la plataforma fue cargado en un arreglo de 64 bits y convertido a un número de señal de 16 bits. Sin embargo, como el número más grande que puede ser almacenado en un arreglo de 16 bits es 32,768 y el número que había resultado de la operación era más grande, la conversión falló, el software no.

El martes 14 de septiembre de 2004 alrededor de las 5 de la tarde, los controladores de tráfico aéreo del suroeste de los Estados Unidos perdieron contacto de radio (voz) con 400 aviones a los que estaban guiando. Los aviones se dirigían en trayectorias que los podrían colisionar, algo que sucede normalmente, pero en este caso no podían comunicarse con los aviones para cambiarles el rumbo. Dentro de la unidad de control hay un reloj que pulsa el tiempo en milisegundos y esta unidad usa dicho tiempo para mandar mensajes. Empieza en el número más grande que el programa puede guardar en la computadora (2^{32}), que es un número más grande que 4 mil millones de milisegundos; cuando el contador llega a cero, el sistema se queda sin reloj y se apaga. Contar de 2^{32} a cero en milisegundos es un poco menos de 50 días, y la agencia

reguladora de vuelos en Estados Unidos, la FAA, hacía que un técnico apagara la máquina cada treinta días, casi tres semanas antes de que se apague sola.

¿Qué paso? La FAA dice que fue un error humano, y yo estoy de acuerdo, del humano que programó el software para que se apagara cada 50 días. ¿Cómo se solucionó el problema?, el de ese día, se resolvió gracias a los teléfonos celulares de los operadores, que buscaron otro centro que pudiera contactar a los aviones. Para el software, se creó una mejora del programa, para que limpiara el contador automática y periódicamente. Yo todavía tengo una duda, ¿qué va a pasar cuando el contador interno de esta nueva mejora llegue a cero?

Sin embargo, no es posible probar todo, una prueba exhaustiva es impráctica, el número total de casos de prueba posibles, para un programa de tamaño moderado es muy grande. La realidad es que, ejecutar toda combinación o todos los casos posibles durante las pruebas es inadecuado, sin embargo, sí es posible cubrir las condiciones lógicas que haga el programa.

¿Qué es un caso de prueba?

Hay muchas definiciones, la más común y que tomaré para este texto es un estándar de la industria, que indica que un caso de pruebas es un conjunto de entradas, condiciones y resultados esperados diseñados con un objetivo particular para verificar un requerimiento específico.

La Teoría de Gráficas es una teoría sólida en matemáticas y las pruebas de software son necesarias, pero ¿tienen alguna relación una con otra?

- Gato de Cheshire, ¿podrías decirme, por favor, qué camino debo seguir para salir de aquí?
- Esto depende en gran parte del sitio al que quieras llegar —dijo el Gato.
- No me importa mucho el sitio… —dijo Alicia.
- Entonces tampoco importa mucho el camino que tomes —dijo el Gato.
- … siempre que llegue a alguna parte —añadió Alicia como explicación.
- ¡Oh, siempre llegarás a alguna parte —aseguró el Gato—, si caminas lo suficiente!

Lewis Carrol, Alicia en el país de las Maravillas

Gráficas y Pruebas

El proceso de desarrollo de software debería llevar implícito el proceso de pruebas. En un proceso completo, las pruebas de software se pueden dividir de muchas maneras, hay una manera básica de dividirlas en dos partes principales, las hechas por el desarrollador de software y las que hace el usuario final del software. Esta separación también es conocida como pruebas de bajo y de alto nivel. Las de bajo nivel son llamadas así, dado que son las primeras que se hacen y generalmente son llevadas a cabo por el desarrollador mismo, siguiendo su programa de cómputo. Dichas pruebas responden a la pregunta de si el software está bien hecho.

Las pruebas de alto nivel las debe hacer el usuario final, antes de que el producto entre en producción, y responden a la pregunta de si el software es el correcto para lo que se necesitaba. Existen, además, otras pruebas más técnicas, como por ejemplo, las pruebas de performance o de comportamiento, que deben de garantizar que el producto final cubra con ciertas especificaciones establecidas desde el principio.

Pruebas de bajo nivel

Estas pruebas generalmente son muy técnicas, por lo que puede parecer un poco tedioso el tratar de verlas con detalle, y frecuentemente requieren del conocimiento de la programación. Watson y McCabe (1996) documentaron una metodología y una medición sobre la complejidad de un programa muy completas para este tipo de pruebas. Trataré de explicar esta metodología en este apartado.

Usaré el lenguaje de programación C para los ejemplos presentados, pero podría usarse cualquier otro.

- Digráficas de flujo

Las digráficas de control de flujo o digráficas de flujo describen la estructura lógica de programas o módulos de software. Un módulo de software es una función simple o una subrutina en un lenguaje de programación, el cual tiene una sola entrada y salida y puede ser llamada por algún proceso; en el lenguaje C un módulo es una función. Cada digráfica de flujo consiste en vértices (nodos) y flechas, donde los vértices representan declaraciones o funciones o expresiones y las flechas la transferencia de control entre los vértices.

Como dato interesante para los programadores, podríamos decir que una digráfica de flujo es, casi, una abstracción de un diagrama de flujo del programa.

Cuando se ejecuta el programa o módulo, se recorre el programa a través de trayectorias en la digráfica de flujo de la entrada a la salida del programa. Toda trayectoria posible de ejecución en un módulo tendrá una trayectoria independiente, donde diremos que dos trayectorias posibles de ejecución, digamos T_i y T_j, son independientes si $V(T_i) \neq V(T_j)$, es decir, trayectorias que pasan por al menos un vértice distinto en la digráfica de flujo. La correspondencia entre las trayectorias posibles de ejecución y las trayectorias de la digráfica de flujo es la parte fundamental de la metodología estructurada de pruebas.

Veremos representado esto en un caso muy sencillo, supongamos que tenemos el siguiente programa que calcula el máximo común divisor (mcd) de dos números enteros positivos:

Programa en C del máximo común divisor

```
v0      mcd (int m, int n)
        { /* Asumimos m y n enteros positivos >0,
        */
        int r;
v1      if (n>m) {
v2              r = m;
v3              m = n;
v4              n = r;
v5              }
v6      r = m % n /* m módulo n */
v7      while (r !=0) {
v8              m = n;
v9              n = r;
v10             r = m % n /* m módulo n */
v11             }
v12     return n;
v13     }
```

Para poder obtener la digráfica de flujo de este programa de cómputo, primero se deben numerar las instrucciones del programa. Así, la digráfica de flujo tendrá como vértices cada una de estas instrucciones (en el lado izquierdo del programa se puede ver la numeración de las instrucciones desde *v0* a *v13*), y las flechas estarán conectando cada una de ellas siguiendo las instrucciones del programa.

En este ejemplo, la ejecución del programa (módulo) empieza en el vértice *v0*, la primera instrucción es el *if* del vértice v1, con la salida de verdadero al *v2* pasando a *v3* y a *v4*, terminando en *v5*; la salida del *if* de falso va directamente del *v1* al *v5*, formando con esto un primer ciclo del programa. Siguiendo las instrucciones del programa llegamos al segundo ciclo en el *while* (*v7*) y, siguiendo todas las instrucciones de la misma manera, llegamos hasta la decisión de salida del programa representada en el vértice *v13*. En la figura 10 está la representación de este programa.

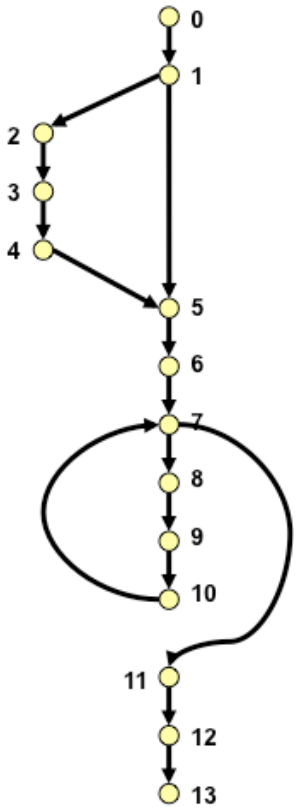


Figura 10. Digráfica de flujo del programa del máximo común divisor.

Esta es la llamada digráfica de flujo del programa del máximo común divisor, en la cual tenemos 14 vértices (numerados de *v0* a *v13*) y 15 flechas. Es una digráfica, ya que hay un flujo que va de arriba hacia abajo y además hay una dirección entre cada uno de los conectores. Es común, dentro de los libros de pruebas, no poner direcciones en las digráficas de flujo y usar el término de gráfica, ya que en las pruebas no se ve la parte teórica que hay detrás y, por lo tanto, no se presta a confusión.

Actualmente ya no se necesita hacer la digráfica de flujo de manera manual para cada programa, existen herramientas en el mercado (la del mismo McCabe por supuesto) que lo hacen automáticamente.

La forma como trabajan estas herramientas es similar al proceso que describo: para cada uno de los programas, la herramienta le pone etiquetas al software y luego lo compila con estas etiquetas, produciendo las digráficas de flujo del programa. Esto simplifica mucho el trabajo, aunado a que la forma como lo hace la herramienta es transparente para el programador o el probador. Sin embargo, es necesario por supuesto entender qué es lo que hacen las herramientas antes de usarlas.

La metodología descrita por McCabe (1989) define, con base en la digráfica de flujo, una medición de complejidad para el software. Medir la complejidad del software ha sido, a través de los años, un tema de estudio muy polémico para los desarrolladores. La manera más sencilla de hacerlo es contar el número de líneas de código, aunque dicha medición no es muy buena para medir complejidad ya que solamente toma en cuenta el tamaño del software. Por ser una medida fácil de calcular, en muchos países es usada para pagar los servicios o programas que se hacen o corrigen, pero realmente no están tomando en cuenta lo "complejo" de un programa; por ejemplo, no es lo mismo usar veinte líneas de código para sumar números, que usar cinco para una función exponencial, pues es más sencillo corregir una suma que una función compleja.

Adrián Costea (2007) estudia y compara distintas mediciones de complejidad en las cuales toma en cuenta una medición de McCabe muy similar a la que vamos a ver, aunque ésta es más completa ya que toma en cuenta el tamaño del programa. En ese estudio se comenta que lo importante es tratar de medir todos los programas de la misma manera. Actualmente, en muchas instalaciones se usa la herramienta de McCabe para medir la complejidad del software y hacer las pruebas de bajo nivel.

- Complejidad ciclomática

La complejidad ciclomática se define para cada programa (o módulo), a través de su digráfica de flujo, de la siguiente forma:

$CC(D) = e - n + 2$ donde e es el número de flechas y n el número de vértices de la digráfica.

Por lo tanto, la complejidad ciclomática del ejemplo del máximo común divisor (figura 10) es 3 (15 flechas - 14 vértices + 2).

¿Qué quiere decir esto?, no es casualidad que se use el término de complejidad ciclomática para un programa de software y que además tenga esa notación. La notación es heredada de la definición de número ciclomático.

Recordemos como se definió el número ciclomático:

$mu = e - n + 1$

Si nos fijamos bien, la diferencia entre mu y CC, es uno. Si ponemos una flecha virtual entre el nodo final y el inicial de la digráfica (poner una flecha de más, como se vio en el ejemplo del problema del cartero chino; ver figura 11), podemos decir que la digráfica de flujo sin la flecha virtual tiene CC = 3 y con la flecha virtual tiene mu = 3.

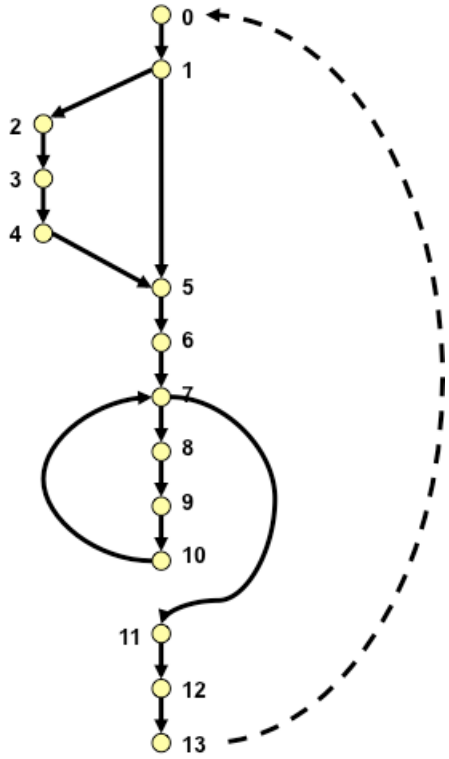


Figura 11. Digráfica de flujo con la flecha virtual.

Con esta flecha virtual podemos "empatar" las dos definiciones y con ello tratar de explicar la complejidad ciclomática de un programa.

La complejidad ciclomática nos dice cuántas flechas se le tienen que quitar a nuestra digráfica virtual para que ya no tenga ciclos (dirigidos y no dirigidos); es claro que la flecha virtual es una de ellas, así fué como la construimos para armar un ciclo, pero las demás flechas que hay que quitar están en los "ciclos" o decisiones que toma el programa.

Para verlo con el ejemplo del programa del mcd, los tres ciclos están definidos así: el primero con el *if* (del vértice 1 al vértice 5 o del vértice 1 al vértice 2), el segundo con el *while* (del vértice 7 al vértice 11 o del vértice 7 al vértice 8) y el tercero que se generaría al poner la flecha virtual (del vértice 13 al vértice 0, ya que seguir todo el flujo y regresar al principio genera un ciclo).

Cuando probamos, lo que buscamos es encontrar el número de trayectorias independientes en la digráfica de flujo, pero este número puede ser muy grande. La complejidad ciclomática es un número que podemos calcular, el cual nos muestra, además, el número de los "ciclos o decisiones" del programa.

Al ser el tiempo dedicado a pruebas, muy corto, lo que buscamos son trayectorias óptimas e independientes que pasen por estos ciclos.

Para apreciar que la complejidad ciclomática se puede además considerar como el número mínimo de trayectorias independientes para probar, es necesario introducir algunos conceptos de álgebra lineal; aunque no ahondaré en la demostración de esto, sí explico un poco al respecto en el apéndice A.

La función que asocia a cada digráfica su complejidad ciclomática será llamada simplemente su complejidad, y es llamada así, ya que si ésta es grande significa que el programa es "complejo" o que tiene muchas decisiones que tomar. Esto puede causar debate entre los programadores, sin embargo, en términos de pruebas lo que quiere decir es que el programa tiene muchos ciclos o preguntas del tipo *if*, *then* o *while*, entre otros y el probar todas estas condiciones puede llevar mucho tiempo. Los puristas de la programación dicen que limitar este número es limitar la libertad de escribir software –'no tiene que ser sencillo para estar bien hecho'-; lo difícil, es probarlo. Si un software tiene muchos ciclos puede ser muy complicado para probarlo o, lo que realmente buscamos, crearle buenas y eficientes trayectorias de prueba, como se verá a continuación.

- Pruebas estructuradas

Una vez que tenemos la complejidad ciclomática de un programa, hay que buscar las trayectorias independientes y hacer una prueba estructurada, es decir, una prueba que ayude a tratar de probar "todo" de una manera rápida y eficiente. Por ello, las pruebas se limitarían a probar el software en esta etapa, con dichas trayectorias.

Hay varios tipos de pruebas de bajo nivel, una de ellas es la prueba por ramas de software, en este tipo de pruebas, se crea un caso de prueba por cada rama o decisión del mismo, veamos un ejemplo de otra función en C.

```
void func (0)

{
        if (condición 1)

                x = x + 1;

        if (condición 2)

                x = x - 1;
}
```

Si probamos con solamente dos casos de prueba y asumimos que las dos condiciones que se prueban (condición 1 y condición 2) producen falso, entonces *x* no sufre modificación (no entra a ninguna condición), si hacemos que las dos condiciones

produzcan el valor verdadero, también *x* resulta no modificada (primero se suma uno y luego se le resta).
De esa forma se pasa por todo el código del programa y sus condiciones probando las dos ramas del programa, sin embargo, la prueba de tener un caso con una condición verdadera y la otra falsa, la cual si produce un cambio en x, no sería probada.

Desde un punto de vista estructurado, no basta con pasar por todo el programa.

Como la complejidad ciclomática de este programa es 3, ya que al tener dos If se crean dos ciclos, al menos una trayectoria de pruebas debe de tener un falso y un verdadero, lo cual hará que *x* no retenga su valor y podamos detectar si hay un error del programa en las pruebas. Ello nos lleva a buscar un método para encontrar estas trayectorias de prueba, que es la base para poder probar estructuradamente usando la técnica de McCabe.

La herramienta de McCabe, además de hacer la digráfica del programa y los cálculos de la complejidad ciclomática, también nos ayuda a crear estas trayectorias de prueba, con lo cual se simplifica mucho la labor del probador.

- Un método para encontrar las trayectorias de prueba

El primer paso para encontrar nuestro conjunto de trayectorias de prueba es partir de la digráfica de flujo, tomando una trayectoria base que trate de cubrir el programa desde la primera instrucción hasta la última. Dicho de otro modo, debemos seguir el programa tratando de recorrerlo completo de principio a fin. Esta elección de trayectoria es en cierto modo arbitraria, y generalmente depende del conocimiento o interés del probador o del programador. Así, la trayectoria base debe de ser la que el probador designe a su juicio como la más representativa del programa o módulo.

Tomando esta trayectoria como base, para generar la segunda trayectoria se debe recorrer la primera y cambiar, en la primera decisión a tomar, a la ruta alterna de la trayectoria (lo que sería equivalente a ir por la otra flecha, en el primer ciclo de la digráfica de flujo); tratando de que la mayoría de las siguientes decisiones sean las mismas que se tomaron en la trayectoria base. De esta manera, las dos trayectorias se reúnen después de la decisión, hasta llegar a la salida del programa.

Para generar la tercera trayectoria, se parte otra vez de la trayectoria base pero ahora se debe variar la segunda decisión, y así se van generando sucesivamente las demás trayectorias hasta cubrir todas las decisiones de la trayectoria base. Una vez hecho esto, tomamos la segunda trayectoria y continuamos el proceso cambiando las decisiones; cuando cubrimos todas las decisiones, nuestro conjunto de trayectorias está completo.

El truco de este método, en el caso de pruebas estructuradas, es recorrer toda la digráfica a través de las decisiones del programa. Como hay tantas decisiones como ciclos se forman en la digráfica, esto implicaría que estamos probando "todo", aunque no es totalmente cierto, pues se dejan funciones sin probar. Dicha diferencia se debería de probar en las pruebas de alto nivel.

Veamos rápidamente un ejemplo para explicar la generación de los casos de prueba y usando la complejidad ciclomática del programa del mcd que vimos al principio. Si partimos de la digráfica de flujo del programa (figura 10) hay que generar tres trayectorias para cubrir todas las decisiones del programa. Es claro que tenemos que conocer el programa, por eso estas pruebas son generalmente hechas por los desarrolladores o por un programa que "lea" el código y nos diga que ruta debemos de tomar, como en el caso de McCabe.

Lo primero es elegir una trayectoria que recorra la digráfica, así que tomaré la trayectoria de dos números que sean "primos relativos", es decir, que su máximo común divisor sea uno, así la trayectoria que seguiría sería la primera de la figura 12. La segunda trayectoria podemos hacerla cambiando la decisión del *if*, podrían ser estos dos mismos números pero invertidos (que de hecho, es lo que hace el programa en el *if*, invertir los números). La tercera trayectoria y última que necesitamos, ya que la complejidad ciclomática es 3, la obtenemos con dos números cuyo mcd sea uno de ellos.

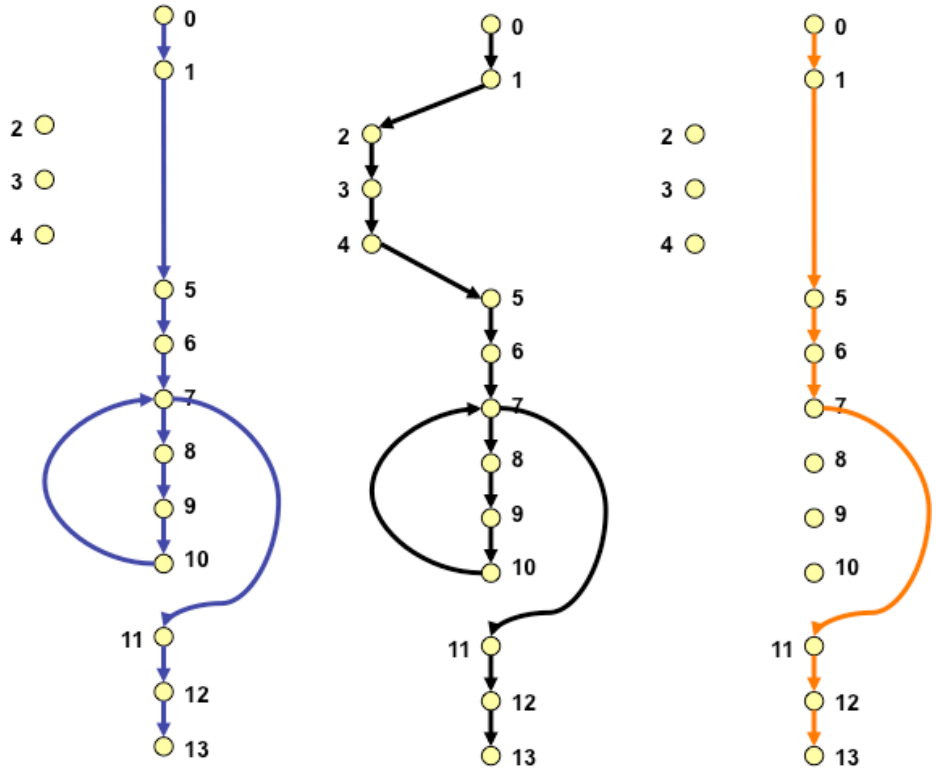


Figura 12. Tres trayectorias independientes del ejemplo de la digráfica del mcd.

Con estas tres trayectorias no cubrimos todo el programa del mcd, pero si podemos hacer una prueba rápida y estructurada sobre él.

Para el primer caso podemos usar, por ejemplo, los números 2 y 5, con lo que tendríamos *n* = 2 y *m* = 5. El programa en estas condiciones no entra al *if* que los cambiaría, por lo que pasa del vértice 1 al 5, luego en el punto 7 entra al *while* ya que 5 módulo 2 da 1 que no es cero; terminando el *while* sale del programa regresando el valor de 1. Análogamente podemos poner datos para los casos dos y tres usando las parejas (7, 2) y (4, 8) respectivamente; en el último de los casos no se entra al *while* del programa, ya que 8 módulo 4 es cero.

Como se ve en este apartado, podemos explicar la Teoría de McCabe con Teoría de Gráficas. Mucha gente no está de acuerdo con esta metodología y recurre a argumentos no muy científicos para desacreditarla. Sin embargo, el tener una metodología y una medición estándar es un avance importante en el proceso de pruebas.

Por ejemplo, un punto en contra dice que la teoría no es "cerrada", por lo que si tenemos un programa ello produce una digráfica de flujo; pero lo contrario no necesariamente es cierto. Una digráfica de flujo puede estar relacionada con varios programas de software o con ninguno, por lo que no se tiene una relación uno a uno.

Además, McCabe asegura que si el número ciclomático es muy grande, la posibilidad de error es más grande, pero no hay datos duros ni teoría que lo sustente, es decir ¡no se ha probado del todo! Eso sí, un programa con un número ciclomático grande puede ser más "complejo" de seguir que un programa sin tantos ciclos, pero en general, los gurús, hackers y programadores profesionales no lo consideran así. Hay casos en que programas que hacen lo mismo pero que fueron programados de manera distinta, producen una complejidad dispar y, finalmente, sabemos que no estamos probando todas las trayectorias posibles, lo cual no sería humanamente posible.

Hay que tomar en cuenta, por otro lado, que un programa con muchos ciclos siempre será más difícil de probar, sobre todo por un tercero, y que toda teoría que ayude a simplificar, estandarizar o mejorar los procesos de prueba, aunque solo sea en parte, es bienvenida.

Pruebas de alto nivel

Una vez que el software "está listo" (o más bien, los programadores ya no quieren saber nada de él), debe probarse y validarse que la construcción del programa en cuestión cumple con lo que se pidió originalmente. En otras palabras, ya tenemos la programación lista pero ahora tenemos que comprobar que haga lo que tiene que hacer; primero validamos que la programación no tenga errores y, lo más importante para el que solicita el software, verificamos que los requerimientos que se pidieron originalmente se cumplan totalmente.

Un ejemplo muy sencillo para explicar este punto sería usando el programa que vimos del máximo común divisor, puede estar bien programado y funcionando bien, pero si la solicitud del usuario fue calcular el mínimo común múltiplo, ese programa claramente no lo hace.

La idea de tener una prueba de alto nivel es validar los requerimientos del usuario, y además que sea eficiente, de manera que lo mejor es que los casos de pruebas también sean eficientes. Para ello, primero tenemos que crear un buen modelo que nos permita entender qué es lo que debemos de probar. El modelado de pruebas es un gran avance en dichos procesos, ya que podemos seguir modelos como el que vimos en las pruebas de bajo nivel y hacer los casos de prueba consecuentemente. Sin embargo, estas pruebas muchas veces dependen de tiempo, de que el software esté terminado y que de algún modo el usuario conozca la programación. Por ello, es recomendable buscar caminos más rápidos o en paralelo al desarrollo que nos permitan hacer un buen modelo de nuestro sistema.

- Modelado de pruebas

Un buen modelo de pruebas se puede dar no a través del código sino a través de la especificación original del software, para crear los casos a la par que se construye el sistema. Podemos desarrollar un modelo que genere, por ejemplo, un árbol de decisión sin tener todavía el software desarrollado y crear así nuestros primeros casos de prueba a través de dicho árbol (árbol, ¿otra coincidencia con gráficas?).

Una vez que tenemos el desarrollo podemos probarlo con los casos que obtenemos del modelo, por ejemplo, si tenemos que probar un nuevo software que se va a instalar en un cajero automático, podemos modelar las transacciones para probarlas con un modelo parecido al de la figura 13, sin tener aún la programación terminada:

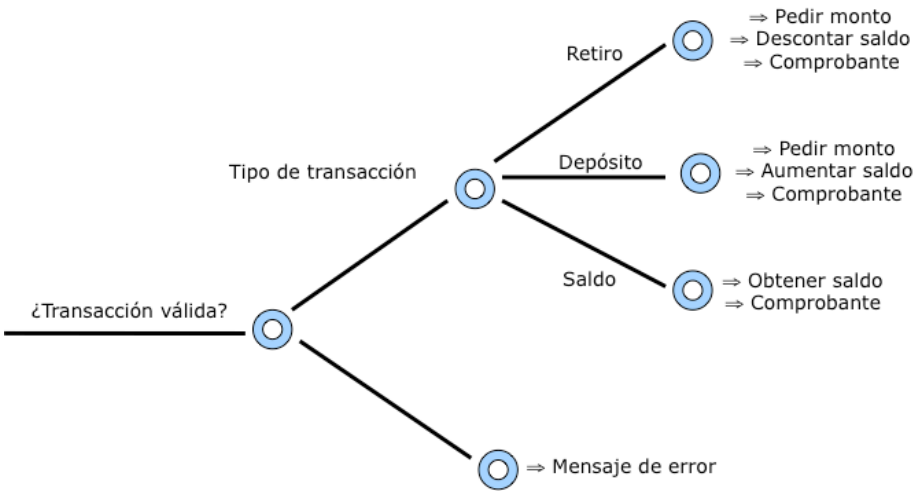


Figura 13. Parte de una transacción de cajero automático vía un árbol.

Así, sabemos qué es lo que debe de hacer el programa. Mientras tanto, el desarrollador debió probar el código con un método similar al que ya vimos. Para crear nuestros casos de prueba, lo que debe hacerse entonces es seguir las trayectorias del árbol y hacer al menos un par de casos de prueba (un caso exitoso y uno fallido) por cada rama del árbol. Por ejemplo, un primer caso sería tener una transacción (tarjeta) válida, pedir una transacción de retiro y pedir un monto de alguna cantidad; este guión es independiente de si tenemos o no el software, una vez que lo tengamos, probaremos que esta transacción sea válida.

Una manera de probar muy usada por los usuarios con este modelo es buscar primero las trayectorias más grandes y posteriormente probarlas de la mayor a la menor; sobre todo si se tiene poco tiempo y no es posible cubrir todos los casos, sólo así es posible cubrir un mayor número de decisiones. Esta técnica tiene un equivalente en Teoría de Gráficas, que es la de encontrar la trayectoria más grande (o el diámetro de un árbol), sin embargo, el algoritmo para obtenerlo es similar al proceso que haría un usuario y, en este caso particular, podría no mejorar el tiempo de la prueba. Lo que puede hacerse es priorizar alguna ruta, ya sea por importancia, por valor de retorno o por alguna otra variable, y encontrar una ruta óptima; en esto también se pueden usar resultados de Teoría de Gráficas. En la sección de otras pruebas veremos un ejemplo que podría aplicarse en este caso, así que seguiré un poco adelante para ver un par de casos interesantes.

La técnica de hacer un árbol de decisión puede ser buena en ejemplos pequeños, pero para programas muy grandes puede resultar ineficiente o muy costoso. Lo que sí nos permite es obtener una buena idea de lo que debe hacerse para probar el modelo.

Otra técnica usada en pruebas por los usuarios es la llamada de caminos aleatorios (a veces llamada de "borracho"), la cual sugiere empezar con un modelo, que podría ser el de árboles de decisión, o por uno de los procesos del sistema. Ya con el modelo, se debe partir de un vértice cualquiera y avanzar aleatoriamente por el mismo; estos casos son sencillos de implementar y curiosamente sí han ayudado en casos exitosos como el de Microsoft, documentado por Nymann en una conferencia de software en 1998, ya que prueban ramas que de otro modo pueden pasar desapercibidas, al no considerarse importantes.

La técnica aleatoria está basada en lo que en ingeniería de sistemas se conoce como la "teoría del chango", la cual supone que si se tiene un "número considerable" de changos y se les pone una máquina de escribir para que tecleen al azar, en un momento dado escribirán algo coherente (a veces se dice una obra como El Quijote, pero eso es más improbable aún). Sin embargo, estas técnicas sólo deben ser utilizadas para complementar pruebas existentes y únicamente si hay tiempo de sobra en el proyecto. Lo interesante de éstas es que asumen que se tiene un modelo, y asimismo se tiene una gráfica, por lo que se pueden utilizar técnicas de Teoría de Gráficas.

El modelado de pruebas representa una enorme ventaja sobre los métodos tradicionales, los cuales consisten básicamente en estudiar qué es lo que hace el software y escribir y ejecutar los casos de prueba sobre esa base. Los modelos tradicionales siempre dejan defectos en la programación, ya que puede haber requerimientos que se piden y no se hacen, pero tampoco se prueban. Además, los casos tradicionales pueden convertirse en lo que se conoce como la paradoja del pesticida (Beizer 1990), ya que al correr los mismos casos sobre el mismo software, los defectos que buscan estos casos ya fueron encontrados desde el principio y no va a encontrarse ningún otro, ya que no será buscado por esa prueba. La palabra pesticida se refiere a que en inglés un defecto se le conoce como "bug" o bicho y la idea es acabar con ellos.

Existen sistemas que miden el comportamiento o estados del software, como por ejemplo los estados que puede tener la tarjeta en el cajero, o los estados válidos de un cliente en el sistema, o el estado de mensajes o transferencias electrónicas enviados entre los bancos. Un ejemplo que no es de software pero puede ayudar a explicar esta idea es tratar de ver las velocidades de un auto automático con el modelo expuesto en la figura 14:

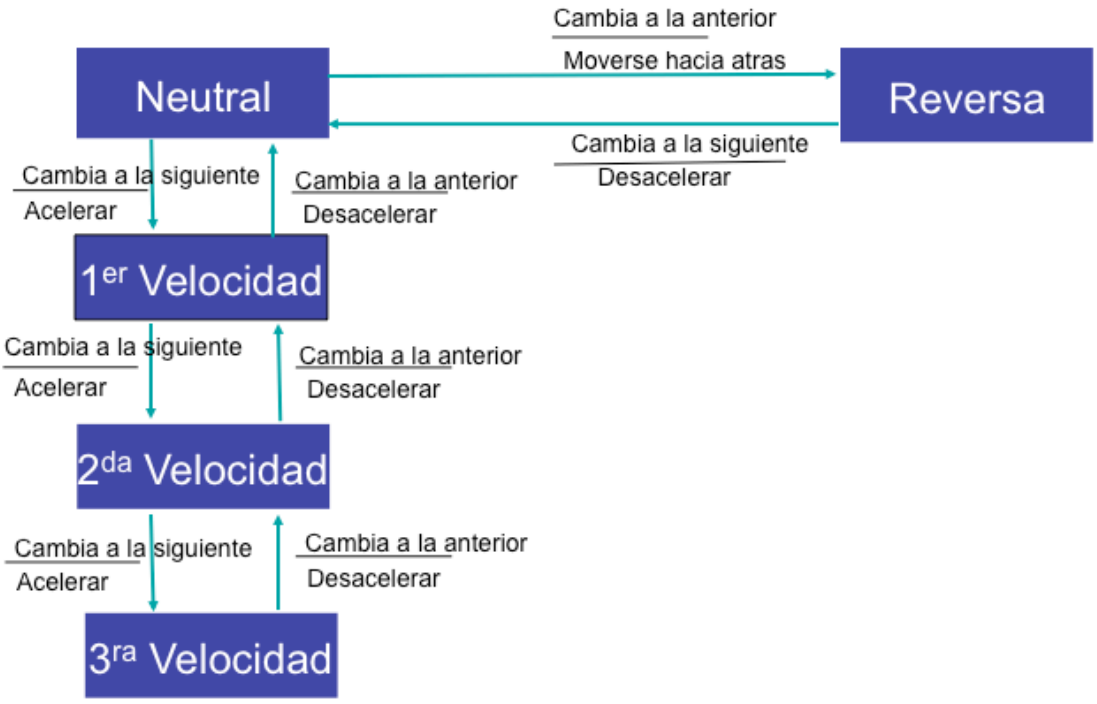


Figura 14. Modelo de velocidades de un auto.

Este es un modelado sencillo y claramente puede verse como una digráfica. Una vez establecido este modelo, se puede hacer fácilmente la prueba a través del mismo; por ejemplo, sabemos que no podemos ir de reversa a primera o segunda y que el auto bajará de tercera a segunda y de ahí a primera. Si lo "complicamos" podríamos hacer el mismo modelo para un auto manual y poner los cruces de tercera y segunda a neutral sin poner los del otro sentido, pero esta es la idea general de un modelo de comportamiento.

Veamos un caso particular e interesante, adaptado para este texto, que encontré en las pruebas del software de un centro de atención telefónica de un banco, donde se tienen cuatro estados de comportamiento, y las relaciones entre ellos, para un nuevo módulo de atención que están probando.

Los cuatro estados son los siguientes: contestar la llamada (C), llamada en espera (A), hablar con el ejecutivo (B) y colgar la llamada o hacer una llamada (D). Mientras que las relaciones se dan de esta manera: de la contestación se puede pasar a hablar con el ejecutivo (c), dejar la llamada en espera (g) o colgar la llamada (f). Si se cuelga la llamada, sólo puede hacerse una nueva llamada (e), y así continúa hasta terminar con el modelo de la figura 15.

Lo que se quiere hacer posteriormente es probar todos los estados posibles en una sola pasada o con un solo caso de prueba, empezando en contestar la llamada.

Figura 15. Modelo y digráfica del problema de comportamiento del Software Telefónico.

Si la vemos con cuidado, adrede la pusimos de una manera similar al de nuestro problema del cartero (figura 8), pero tiene una diferencia, aquí las aristas sí tienen una dirección lo que la convierte en una digráfica. Repasando un poco la historia podemos ver la solución a este problema.

En 1983, algunos años después del problema del cartero chino, Bodin y Tucker presentan una variación de este problema pensando en qué pasaría si las aristas tuvieran una dirección, lo que ejemplificaron con los barredores de la ciudad de Nueva York: éstos, al tener barredoras automáticas, no podían circular por las calles en sentido contrario y la mayoría de las calles son de un solo sentido, o solamente se permite circular de un lado.

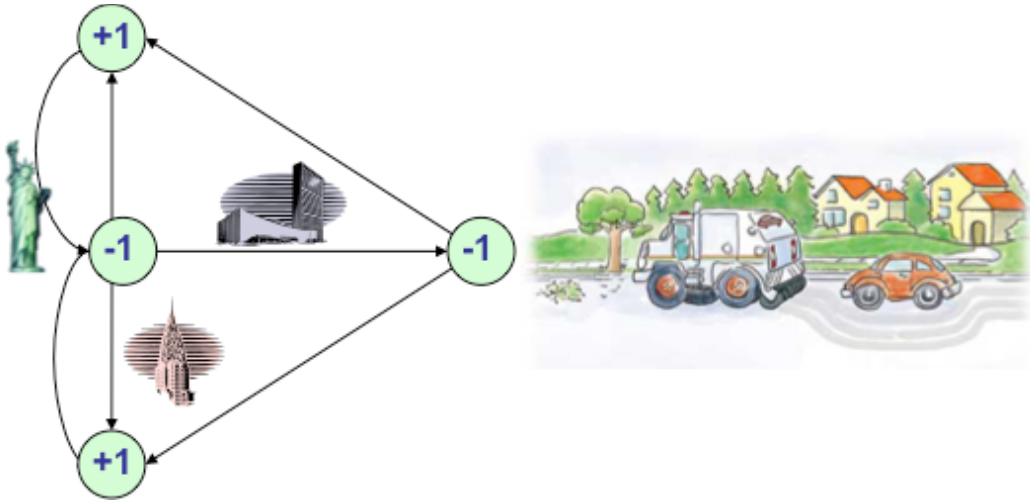


Figura 16. Barredora en las calles de Nueva York (dibujos tomados de internet).

Para construir un paseo Euleriano en el caso del cartero chino, el problema consistió en resolver el número de aristas que tocan cada vértice (grado del vértice). En el caso de una digráfica, además hay que considerar la dirección de las flechas, lo que implica que no solamente se debe de tener un número par de flechas, se debe de tener el mismo número de flechas de entrada que de salida.

Podemos definir a la polaridad de cada vértice como $p(v) = \delta^{+}D(v) - \delta^{-}D(v)$, la diferencia del exgrado y el ingrado, o la diferencia entre el número de flechas de salida menos el número de flechas de entrada; si la polaridad del vértice es positiva, querría decir que tiene más flechas de salida que de entrada, si la polaridad es negativa, querría decir que tiene mas flechas de entrada y, si la polaridad es cero, tendría el mismo número de flechas de entrada que de salida. En la figura 16 se incluye la polaridad de cada vértice.

En la figura 17 utilicé el mismo nombre de las aristas de nuestro problema inicial de la figura 15 y dupliqué los flechas *b*, *e* y *g* para balancear esta digráfica, este es el método que utilizaron Bodin y Tucker duplicando ciertas flechas para tener una polaridad de cero en cada vértice.

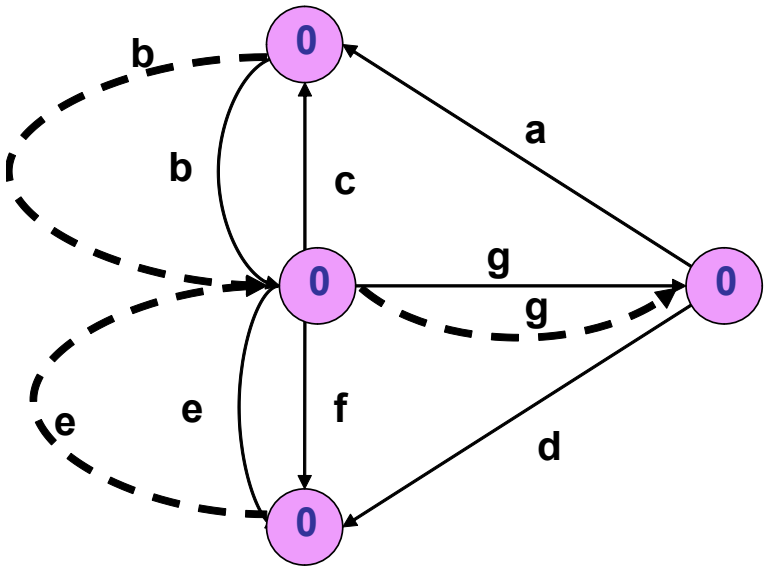


Figura 17. Eulerización del modelo de la barredora y del software telefónico.

Ahora, podríamos recorrer la digráfica o hacer nuestro guión de la prueba para recorrer todos los puntos con una sola pasada, de esta manera:

e g d e g a b c b f

y solamente repetimos en la ruta los segmentos que adicionamos.

En este caso, el guión de prueba que se usa, visto desde la perspectiva del agente para probar el software de atención telefónica es de la siguiente manera: primero el cliente marca al sistema (e), el sistema o el agente dejará al cliente en espera (g), se cortará la llamada o alguien colgará (d), el cliente vuelve a marcar (e), el agente o sistema lo deja en espera (g), el agente recibe la señal de atención (a), el agente habla con el cliente (b), el agente sigue en atención (c), contestará para finalizar la llamada con el cliente (b) y finalmente, colgarán y terminará la llamada (f). Si nos fijamos bien, con un solo caso de prueba recorrimos todas las opciones de este paquete que era precisamente lo que buscábamos. Del mismo modo, podemos pensar que la barredora usará el camino encontrado para poder recorrer las calles de Nueva York en el sentido correcto y solamente pasando dos veces por las calles b, g y e.

Concluyendo este apartado, para una prueba de alto nivel lo ideal es modelar el sistema que se tiene que probar, lo cual se puede hacer aún sin tener la programación completa, ya que el objetivo es encontrar las rutas de prueba. Dicho de otro modo, modelar un sistema nos permite modelar las pruebas necesarias, independientemente de la programación, y es la base de una prueba de alto nivel.

El 3 de enero de 1999 fue lanzado al espacio el Mars Polar Lander (MPL) un módulo espacial diseñado para llegar a estudiar el planeta Marte. El 3 de diciembre de 1999 se perdió al tratar de aterrizar en Marte. Blackburn (2001) documenta, con un modelo de pruebas, que el MPL se perdió en Marte por una falla en el software de aterrizaje, debida a un error en la interpretación de un requerimiento, lo cual pudo resolver haciendo un modelo de pruebas.

El modelar un sistema permite usar técnicas y resultados de la Teoría de Gráficas para simplificar o mejorar las pruebas. Se pueden complicar más y más los casos que hemos visto, pero creo que hasta aquí está suficientemente expuesto este punto. Para los amantes de la Teoría de Gráficas, en el apéndice B incluí una complicación muy interesante del problema de la barredora.

Otro tipo de pruebas

Como hemos platicado, hay mucha literatura sobre pruebas y los diferentes tipos de pruebas que existen. Así, el hacer pruebas es elaborar un proyecto completo dentro del proyecto general del desarrollo del software. Las pruebas en sí también tienen un ciclo, hay que planearlas, prepararlas, ejecutarlas, y luego medir el resultado de las mismas. El definir y poner un ambiente de pruebas también forma parte del ciclo de pruebas, que es fundamental dentro de un proyecto, ya que generalmente las pruebas no se hacen en los ambientes de producción.

Un tipo de pruebas que se usa frecuentemente en la industria del software es el de pruebas de *performance*, las cuales "miden" el comportamiento del software. Una vez que decidimos el software que queremos y éste está bien hecho, hay que evaluar su respuesta en términos del tiempo (y/o de las cargas de volumen de datos o de accesos) que deseamos. Por ejemplo, cuando vamos a un cajero y éste tarda más de 10 segundos en mandarnos una respuesta, nos preocupamos y pensamos que el sistema esta "lento", pero si hacemos cola en un aeropuerto y nos dan nuestros boletos y asientos en 10 segundos podemos decir justo lo contrario. Para definir bien estos parámetros, deben pensarse y escribirse desde el principio, aunque se midan hasta el final y después de saber que el sistema está bien hecho.

Es básico definir correctamente las expectativas del sistema en las pruebas, ya que si no se sabe o define a dónde se quiere ir no importará cuál camino se tome. Por ejemplo, en un sistema de red web necesitamos saber un par de cosas: la carga de los segmentos de red por los que pasa el software y el tiempo de respuesta que esperamos en estos segmentos para las cargas predefinidas. No es lo mismo hacer un software para diez usuarios que para un millón, o pasar miles de datos por un cable de teléfono que por uno de fibra óptica.

En nuestro siguiente ejemplo tenemos el caso del software de una empresa que quiere construir una red de pruebas. La empresa tiene una red central en la ciudad de México, con nodos en Monterrey, Guadalajara, Hermosillo y Mérida, y se quiere construir una subred de pruebas en una dichas ciudades, pero el presupuesto para su construcción es muy limitado. En este caso no se puede crear una red de pruebas del mismo tamaño que el de la red de producción, entonces se busca una red que cumpla ciertos parámetros y sirva en la pruebas del software.

Lo que haremos primero es una gráfica de la red donde los vértices sean las ciudades a conectar; en las aristas que los unen ponemos los parámetros de costo de construcción. Para este problema usamos el parámetro de costo, sin embargo, fácilmente podría remplazarse por la carga de la red, por la distancia o por cualquier otro.

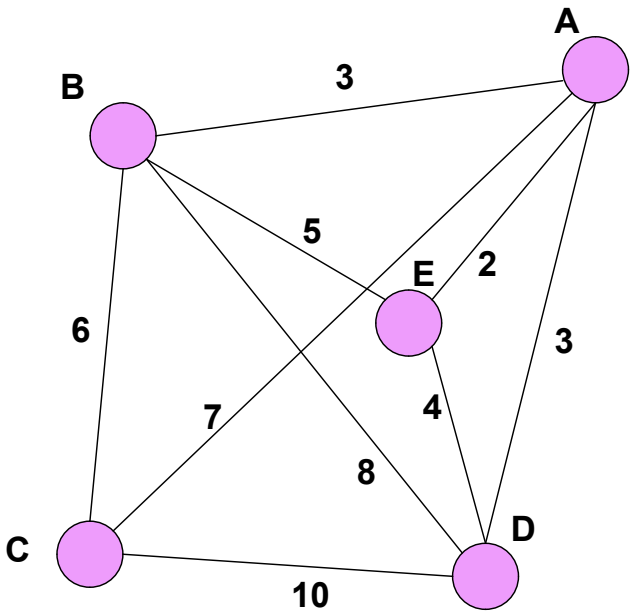


Figura 18. Red de cómputo

La figura 18 muestra la red de cómputo, donde los vértices A-D son centros de cómputo remotos y el vértice E es el punto central de cómputo (en la ciudad de México). Todas las conexiones están dadas en el modelo tal cual están en producción. Dado que se requiere construir la red de pruebas, el costo por construir cada conexión está incluido en la gráfica. Si el objetivo es construir la red más barata, ¿cómo recomendamos la mejor?

Una gráfica S es subgráfica de G, si el conjunto de los vértices de S es un subconjunto de los vértices de G, y el conjunto de las aristas de S es un subconjunto de las aristas de G, es decir: $V(S) \subseteq V(G)$ y $A(S) \subseteq A(G)$.

Si la red la vemos como una gráfica con peso (en este caso el costo), este ejemplo se puede considerar como el caso de un problema general de la Teoría de Gráficas, es decir, si tenemos una gráfica con valor en las aristas, ¿cómo encontramos una subgráfica de ésta tal que sus aristas tengan un valor mínimo y que sea conexa, es decir, que todos sus vértices estén conectados?

Este problema se conoce como el árbol expandido mínimo (AEM), es decir, un árbol dentro de la gráfica cuyas aristas tengan el valor más pequeño posible. Si seguimos la ruta A-B, B-C, C-D, D-E, E-A, formamos un ciclo que une todos los puntos, pero es claro que hay otras estructuras que hacen lo mismo (unir todos los puntos) y más barato, como por ejemplo quitar cualquier arista de este ciclo. Entonces, la pregunta sigue abierta ¿cómo encontramos la más barata? Para encontrar este árbol usaremos el algoritmo de Kruskal.

- Algoritmo de Kruskal

Este algoritmo encuentra el árbol expandido mínimo (AEM) en una gráfica conexa con pesos en las aristas, y le debe el nombre al matemático norteamericano Joseph Kruskal (1928) de la universidad de Princeton, quien lo publicó en 1956 en los Proceedings of the American Mathematical Society.

En términos sencillos el algoritmo hace lo siguiente:

Para una gráfica con *n* vértices adiciona la arista más baja (en costo), evitando la creación de ciclos, hasta que haya adicionado *n* -1 aristas.

Para nuestro ejemplo y empezando con la arista de valor más bajo, tenemos lo siguiente:

Arista	Costo	Decisión
-----	---	----------
A-E	2	Adicionarla al árbol
A-B	3	Adicionarla al árbol
A-D	3	Adicionarla al árbol
D-E	4	Rechazarla forma un ciclo
D-B	5	Rechazarla forma un ciclo
B-C	6	Adicionarla al árbol

Alto - ya que tenemos 4 aristas y la gráfica tiene 5 vértices. Así, el árbol (AEM) generado para nuestra red de pruebas es el siguiente:

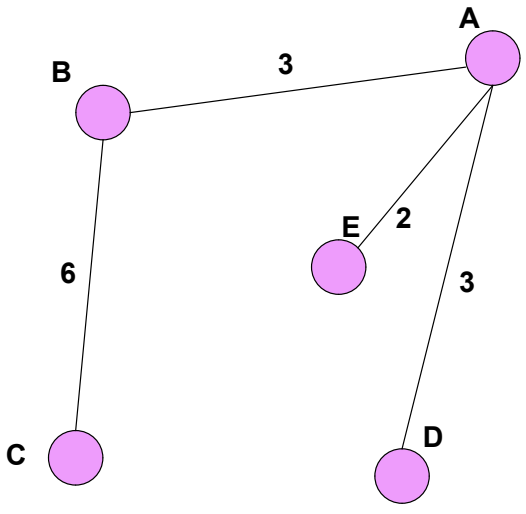


Figura 19. AEM árbol generado con el algoritmo de Kruskal.

Hay que considerar que puede suceder que más de una arista tenga el mismo valor o costo (como en los pasos 2 y 3 de nuestro ejemplo). En este caso, el orden de tomar una u otra no es importante, pero ello podría generar otro árbol (AEM), que sin embargo también tendrá un costo mínimo.

Existen en el mercado muchas herramientas que aplican el algoritmo de Kruskal para redes más complejas. Una aplicación muy importante y pionera de este algoritmo se debe a H. Frank y T. Frisch (1971), quienes diseñaron una red de gasoductos en el Golfo de México. Los ahorros obtenidos por su diseño fueron muy altos, lo que nos demuestra que problemas teóricos de gráficas, como el del AEM, pueden utilizarse para resolver problemas prácticos en muchos campos, además de problemas de software.

Al hablar de redes de cómputo siempre generamos, al menos en la mente, un modelo gráfico de puntos y rayas, donde los puntos representan máquinas y las rayas las conexiones entre ellas. Así, como hemos visto hasta ahora, la Teoría de Gráficas nos ayuda a entenderlo y a resolver problemas.

Seguridad de redes de cómputo

Cuando hablamos de redes de cómputo debemos de pensar en seguridad, ya que siempre se les asocia con palabras como virus, ataques, gusanos, entre otros. En las películas siempre hay un genio malvado que puede entrar a cualquier red o máquina sin importar el lenguaje o el equipo y destrozar al mundo. Por supuesto, hay que probar la seguridad de una red.

En el laboratorio de Virología y Criptología (ESAT) de Francia, un equipo de científicos dirigido por Eric Filiol (2007), con apoyo de la Armada Francesa ESCANSIC, usaron un algoritmo de cobertura de vértices diseñado por Dharwadker (2006) para simular la propagación de gusanos en redes de cómputo grandes. Diseñaron además estrategias óptimas para proteger la red en contra de ataques en tiempo real.

Una cobertura de vértices de una gráfica G es un subconjunto C del conjunto de vértices de la gráfica, tal que cada una de las aristas de la gráfica incide en al menos un vértice de este subconjunto. Se dice que el conjunto C cubre a la gráfica G (ver figura 20).

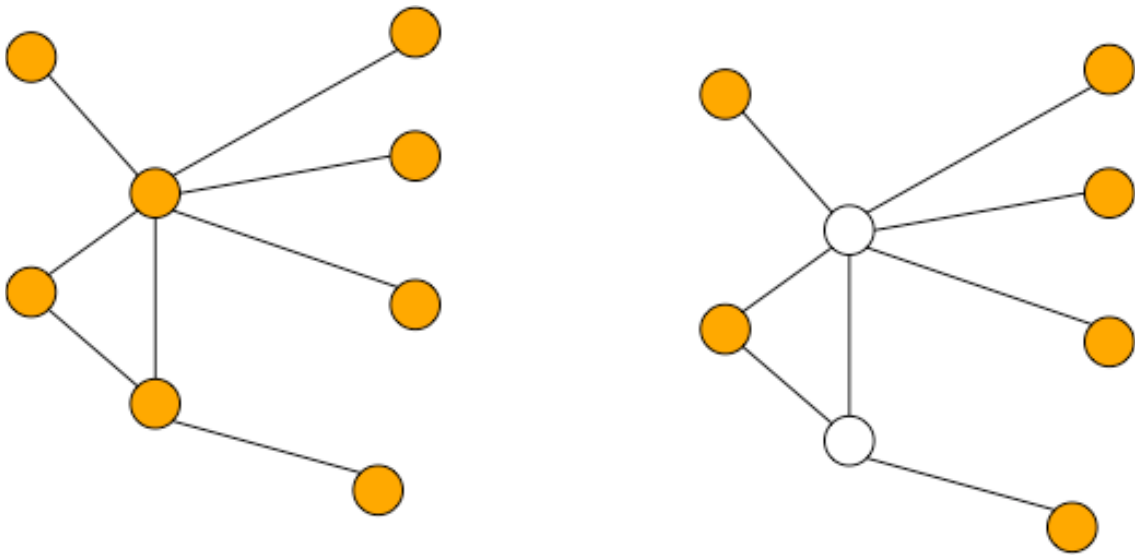


Figura 20. Una gráfica y su cobertura mínima (los puntos blancos serían un conjunto que cubre a la gráfica)

El problema de encontrar la cobertura mínima para una gráfica (o el conjunto C más pequeño), es un problema abierto y uno de optimización en ingeniería computacional. Trabajos como el de Dharwadker (2006) ayudan a resolver problemas particulares como éste.

La simulación que hicieron fue a través de un modelo virtual de red tipo Internet, y demostraron que la topología del ruteo en la red podría tener un impacto en la propagación de gusanos, por lo que algunos servidores de la red juegan un papel más importante que otros.

La capacidad de identificar en tiempo real estos servidores es esencial para obstaculizar la propagación de un gusano. La idea que usaron fue la de encontrar una cobertura mínima de vértices en una gráfica cuyos vértices fueran los servidores y cuyas aristas fueran las conexiones, posiblemente dinámicas, entre los servidores. Ésta fue una solución óptima para diseñar la estrategia de defensa. Si vemos la figura 20 como una red de sistemas, se puede observar que con los dos servidores de la cobertura mínima podemos cubrir la red y partimos de éstos para defenderla; eso mismo hicieron Filiol y su equipo.

No es casualidad que terminemos este texto con un ejemplo de coloración, regresando a la historia, en matemáticas y en particular en la Teoría de Gráficas, donde un teorema famoso es el de los cuatro colores.

- Teorema de los cuatro colores

El teorema de los cuatro colores dice que cualquier mapa en un plano puede ser coloreado usando solamente cuatro colores, de tal manera que regiones o países que compartan o tengan una frontera en común (que no sea un solo punto), no compartan el mismo color.

Este problema fue conjeturado por Francis Gutlirie en 1852, cuando trataba de pintar las regiones de Inglaterra en un mapa, notando que con cuatro colores le era suficiente. Se escribieron muchas pruebas erróneas de esta conjetura, la más famosa por Alfred Kempe en 1879, la cual anunció en Nature y publicó en el American Journal of Mathematics. Francis Gutlirie se volvió famoso e incluso lo nombraron *Caballero*. No obstante, el teorema volvió a ser una conjetura, la de los cuatro colores, cuando Percy Heawood probó en 1890 que la demostración de Kempe era errónea. Así, este problema se volvió una obsesión y resolverlo se convirtió, para algunos matemáticos, en una carrera de vida. No fue sino hasta 1976 en la Universidad de Illinois donde Kenneth Appel y Wolfgang Haken probaron este teorema con la ayuda de computadoras.

Este fue el primer teorema importante de Teoría de Gráficas (y probablemente de matemáticas) que fue probado usando computadoras con algoritmos y procedimientos complejos. El software que usaron seguramente estaba ya bien probado, sin embargo hubo muchos matemáticos que dudaron de la prueba pues no podía verificarse manualmente.

Poco a poco la comunidad fue aceptando la prueba, la cual ocupa casi un libro completo, y detalles de la misma aparecen en un artículo muy interesante del Scientific American (1977). Actualmente, los algoritmos que se usaron ya han sido mejorados y demuestran que el resultado es correcto, sin embargo, el quid de la pregunta sigue siendo el mismo para muchos matemáticos, ¿puede un argumento, que no puede ser revisado directamente por un miembro de la comunidad científica, ser considerado una prueba válida?

La definición formal de coloración en Teoría de Gráficas es la siguiente: dada una gráfica *G* y un número *k* de colores, decimos que una gráfica es *k* - coloreable si cada vértice de G puede tener uno de los *k* colores, y además que vértices adyacentes tengan distinto color. El mínimo valor para *k*, tal que esta coloración existe, es llamado el número cromático de *G* denotado por $\chi(G)$.

Un ejemplo sencillo es el de la siguiente figura, donde tenemos un mapa de México pintado con cuatro colores y una gráfica donde por cada Estado ponemos un vértice y, si hay una frontera común entre ellos, ponemos una arista y esta gráfica la coloreamos con los mismos colores del mapa (figura 21).

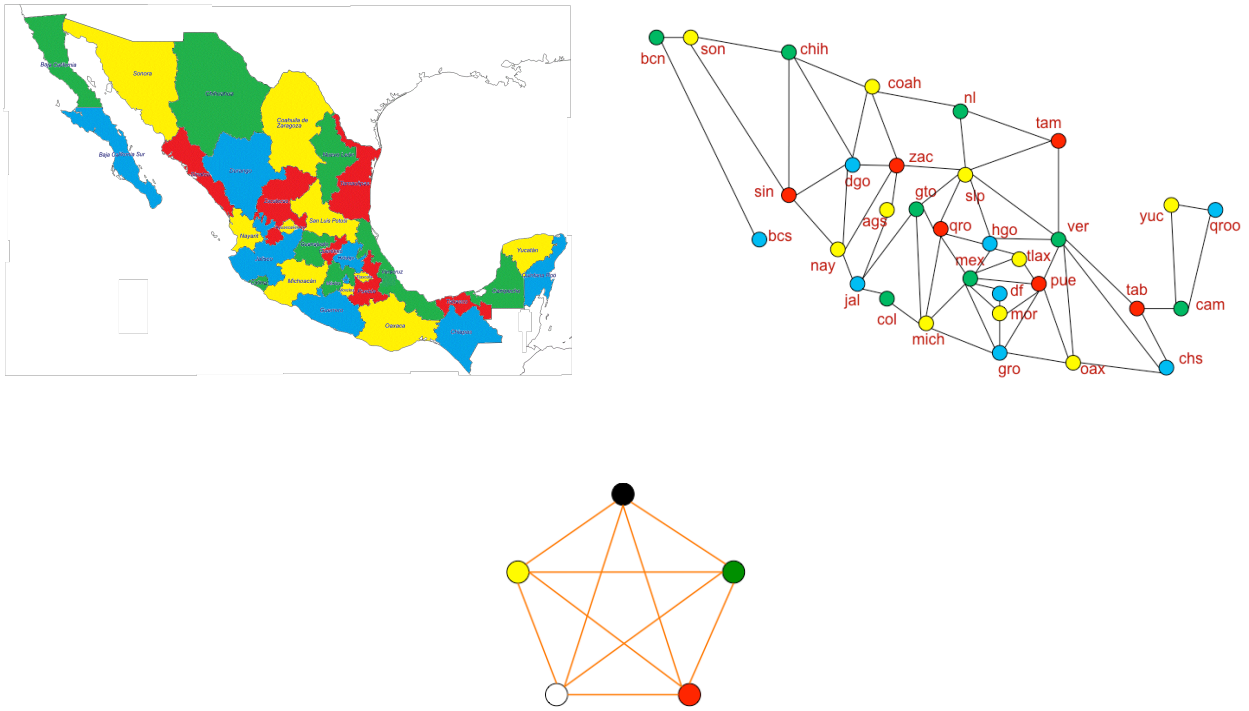


Figura 21. Mapa de México y una gráfica representativa, coloreadas con cuatro colores y la gráfica K_5 coloreada con cinco colores.

Hay que tomar en cuenta que solamente un caso especial de gráficas, las gráficas planas, se pueden colorear con cuatro colores. La gráfica K_5 (ver figura 5b) no puede colorearse con cuatro colores, requiere de cinco; asimismo, K_n, requiere de n colores.

Ahora veamos un problema de coloración en un ejemplo de pruebas de software: durante el día en un banco se van capturando los movimientos de las cuentas de sus clientes, mientras que por la noche se tienen que acomodar los procesos de actualización en el equipo central. Se sabe que ciertos procesos no pueden correr al mismo tiempo (el de cheques con el de tarjetas o el de clientes con el cálculo de impuestos, por ejemplo), por lo que se busca un procedimiento de ordenamiento, evitando así los conflictos que pudieran darse. Es más, se pueden correr todos uno seguido de otro, pero la idea es correr en el menor tiempo posible y comprobar que ello sea posible sin tener que recurrir a pruebas de ensayo y error.

Podemos asumir nuestro problema de ordenamiento como un problema de coloración. Para crear la gráfica se define cada vértice como un proceso (cheques, tarjetas, impuestos, etc.) y se pone una arista entre dos vértices si NO pueden correr al mismo tiempo, es decir, si tienen un conflicto entre ellos. Veamos esto en la figura 22:

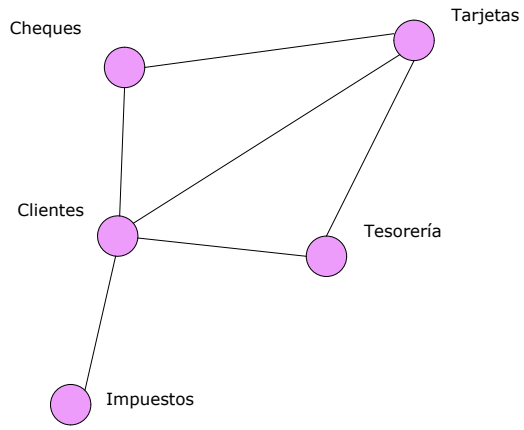


Figura 22. Ordenamiento de procesos

Si tenemos que ordenar los procesos nocturnos de cheques, tarjetas, clientes, tesorería y el de impuestos, la gráfica nos dice que el proceso de tarjetas no puede correr en paralelo al proceso de cheques; pero también nos dice que sí es posible correr impuestos y tesorería al mismo tiempo, ya que no hay una arista (conflicto) entre ellos. Ahora, identifiquemos cada uno de los procesos con un color como en la figura 23 a.

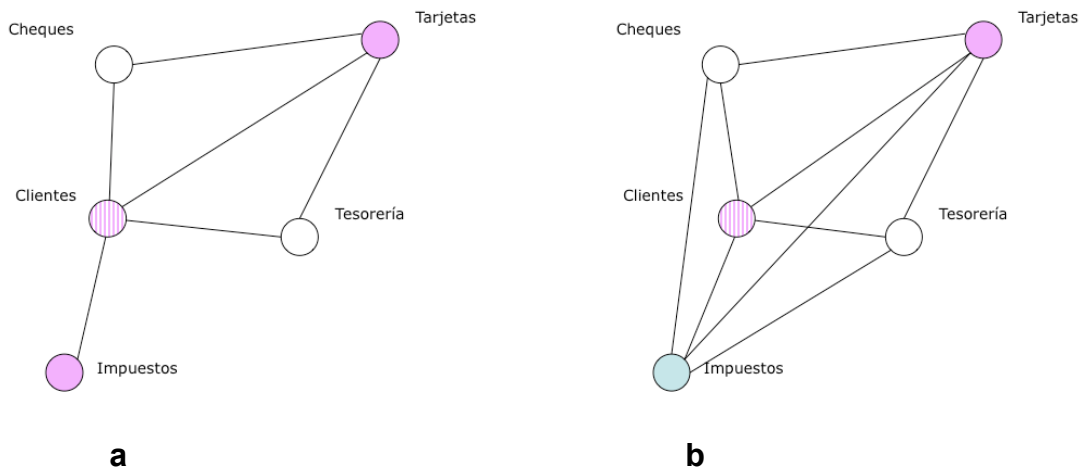


Figura 23. Coloreado de los procesos

Para nuestro ejemplo, con tres colores es suficiente, lo que significa que con tres corridas separadas los procesos nocturnos pueden terminar sin conflicto.

¿Podemos usar menos colores?, en el caso de nuestro ejemplo no, ya que tenemos un triángulo en la figura, por lo que todos los vértices de nuestro triángulo deben de tener un color distinto. En el caso de que el proceso de impuestos fuera dependiente de todos los demás, deberíamos de modificar la gráfica y necesitaríamos cuatro colores como en la figura 23b.

En general, tratar de ver cuál es el número cromático de una gráfica puede ser muy complejo y no se tienen algoritmos rápidos y óptimos para encontrarlos, por lo que sigue siendo un problema interesante y muy estudiado por los matemáticos. En muchos casos, sin embargo, podemos obtener una cota superior de los colores que necesitamos.

Los problemas de coloración aparecen en muchas aplicaciones, lo que me permite terminar con un caso muy interesante: la empresa Akamai Technologies de Cambrige Massachusetts es una compañía de software que tiene una plataforma de cómputo distribuida en 20,000 máquinas por todo Norteamérica. Frecuentemente liberan versiones nuevas del software que distribuyen; dichas actualizaciones no pueden hacerse al mismo tiempo ya que los servidores (máquinas) tienen que apagarse para actualizar el software. Asimismo, no se puede hacer uno por uno ya que llevaría mucho tiempo (el proceso dura aproximadamente una hora), más aún, ciertas máquinas no pueden apagarse al mismo tiempo ya que tienen funciones complementarias o de respaldo.

Este problema lo resolvió la empresa haciendo una gráfica con 20,000 vértices y poniendo aristas donde hubiera un conflicto; la colorearon con ocho colores, de tal manera que solamente se necesitaron ocho rondas de instalación remota para todos los equipos (Akamai es una palabra Hawaiana que significa inteligente).

Do not worry about your difficulties in Mathematics. I can assure you mine are still greater.
Albert Einstein

Un problema abierto

Sabemos que no podemos probar todo, el tiempo para pruebas es finito, lo que debemos hacer es probar lo que debemos y hacerlo bien. Pero si tenemos computadoras, éstas nos pueden ayudar a resolver más rápido cualquier cosa, incluso las pruebas, ¿o no?

Supongamos que las soluciones a un problema pueden ser verificadas rápidamente. Entonces las soluciones ¿pueden encontrarse fácilmente? O dicho de otro modo: si un problema puede ser verificado con una computadora, entonces ¿será posible resolverlo con una computadora?

El problema del agente viajero (o TSP por sus siglas en inglés del Travelling Salesman Problem), es un problema muy estudiado en matemáticas y computación, que básicamente se describe como sigue: tenemos una colección de ciudades y el costo de viajar entre cualquier par de ellas, de manera que el problema del agente viajero es encontrar el camino más barato para visitar todas las ciudades una sola vez y regresar al punto de partida.

Nota: Para efectos de esta presentación se asume que el costo es simétrico, lo mismo que la distancia, es decir ir de *X* a *Y* cuesta lo mismo que ir de *Y* a *X*.

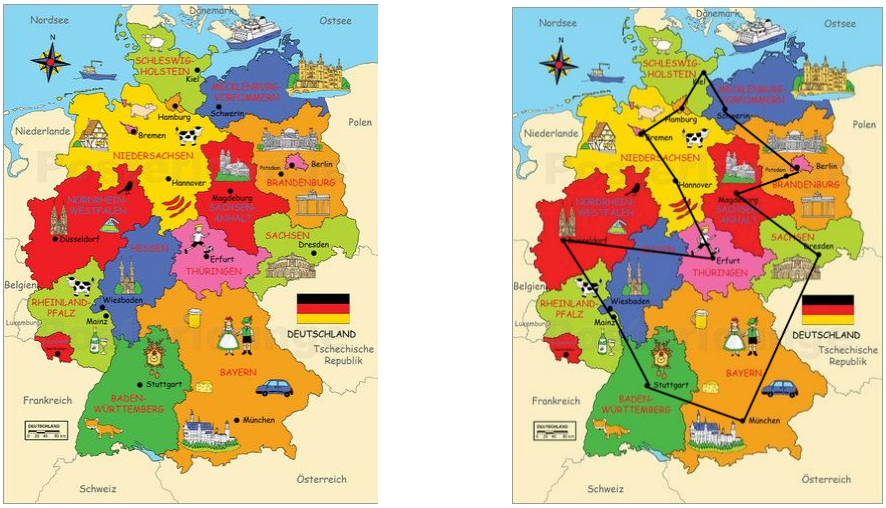


Figura 24. Un mapa de Alemania y un ciclo que pasa por las 15 ciudades más importantes (mapa tomado de internet).

En la figura 24 vemos un mapa con las 15 ciudades más importantes de Alemania, lo que queremos es encontrar un ciclo óptimo que pase por estas 15 ciudades una sola vez. Para ello, hagamos lo mismo que Euler y simulemos el problema con una gráfica, donde el conjunto de vértices representa a las ciudades y una arista es la unión de dos ciudades que estén comunicadas entre sí, y le asignamos el valor del costo. Lo que debemos encontrar en esta gráfica es un ciclo que pase por todos los vértices (ciudades) una sola vez, lo cual es similar al problema original que enfrentó Euler, sin embargo, no es igual.

Recordemos que un paseo Euleriano es un paseo que pasa por todas las aristas de la gráfica y, lo que estamos buscando en este caso, es un ciclo que pase por todos los vértices de la gráfica.

Un ciclo que pasa por todos los vértices de una gráfica, solamente una vez, se le conoce como ciclo Hamiltoniano en honor al científico irlandés William R. Hamilton (1805-1865). En 1859 Hamilton inventó un juego que consiste en encontrar un recorrido por todos los vértices de un dodecaedro (un sólido regular de 12 caras pentagonales), poniendo el nombre de una ciudad en cada una de las esquinas y siguiendo las aristas del dodecaedro (como si fuera un planeta) visitando cada ciudad exactamente una vez.

Se dice que Hamilton vendió este juego en 25 guineas (cuyo valor era un poco mayor a una libra) a un fabricante de juguetes de Dublín.

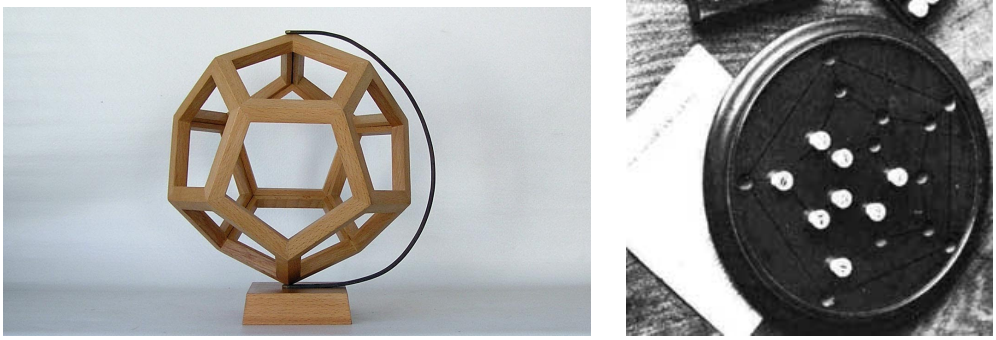


Figura 25. Dodecaedro simulando el planeta y un juego de Hamilton (tomadas de internet).

A una gráfica que tiene un ciclo Hamiltoniano la llamamos gráfica Hamiltoniana. No existe un algoritmo eficiente para saber si una gráfica es Hamiltoniana o no, he ahí el verdadero problema, ¿qué significa eficiente?

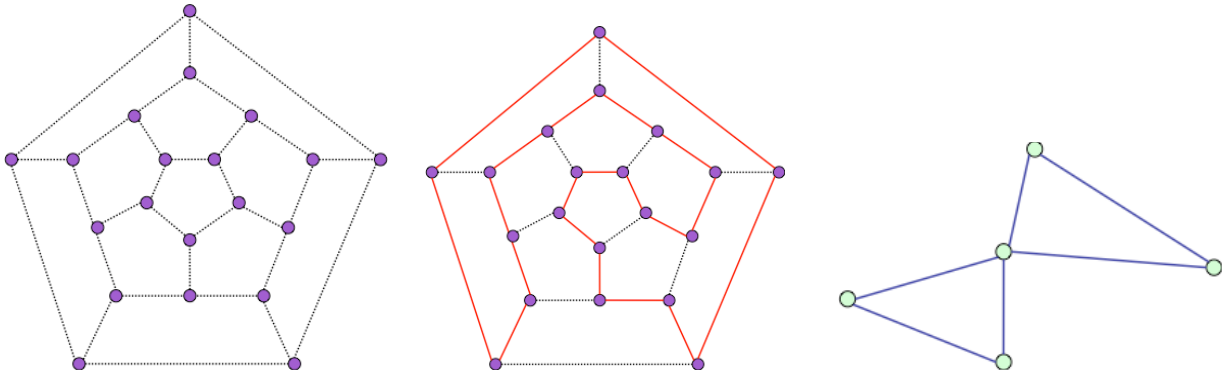


Figura 26. Gráfica del dodecaedro, con un ciclo Hamiltoniano y una gráfica que no puede tener un ciclo Hamiltoniano.

Para resolver el juego del dodecaedro de Hamilton podemos crear una gráfica que lo simule, como en la figura 26, y tratar de construir un ciclo que pase por todos los vértices una sola vez. No existe una forma "rápida" de obtener este ciclo, la única manera conocida es empezar en un punto e ir recorriendo la gráfica tratando de no repetir vértices, y regresar a un punto anterior si ya no hay más opciones, es decir, si no lo encontramos "a la primera", tenemos que cubrir todas las posibilidades.

Una manera para encontrar un ciclo Hamiltoniano en una gráfica como el del problema del agente viajero de la figura 24 sería, por ejemplo: partiendo de una ciudad, revisamos el costo de viajar a una de las siguientes 14 ciudades, de ahí revisamos las restantes, que son 13, luego 12 y así sucesivamente. Pero este método es del orden de 14!, el cual nos da un número extremadamente grande, tendríamos que revisar más de 87 mil millones de combinaciones.

- Algoritmos polinomiales
¿P = NP?

Si un problema lo podemos resolver con un algoritmo de la forma $O(n^k)$, donde k es una constante, decimos que es un problema de complejidad polinomial (P), es decir, se puede resolver “rápido”. En términos de cómputo decimos que lo podemos resolver en un tiempo razonable y medible, es decir de una manera eficiente. Por otro lado, esto no es posible con un algoritmo exponencial o no polinomial (NP), ya que aunque usáramos computadoras, el algoritmo crece tan rápido que no hay máquinas lo suficientemente grandes que lo puedan resolver en un tiempo razonable, generalmente son variaciones de una búsqueda exhaustiva o ataques de "fuerza bruta".

Ejemplos de problemas polinomiales (P) resueltos en tiempo polinomial

- Encontrar la trayectoria más corta entre dos vértices
- Encontrar el árbol expandido mínimo
- Probar que una gráfica es Euleriana

Ejemplos de problemas que no son polinomiales (NP)

- Encontrar el número cromático de una gráfica
- Encontrar un ciclo Hamiltoniano en una gráfica
- Resolver el problema del agente viajero

No existe una condición general para encontrar un ciclo Hamiltoniano en una gráfica, y tampoco existen algoritmos en tiempo polinomial para encontrarlos de manera general.

Sin embargo, una vez que tenemos un ciclo, sí podemos probar en tiempo polinomial que cumple con lo que pedimos. En términos de cómputo, podemos decir que la solución si puede ser verificada rápidamente, aunque dicha solución no se pueda encontrar rápidamente.

El que no se hayan encontrado algoritmos en tiempo polinomial para resolver estos problemas no significa que no existan. Es más, el encontrar un algoritmo para estos problemas o demostrar que no puede existir un algoritmo polinomial es uno de los “problemas del Milenio” del Instituto de Matemáticas Clay (CMI); y a quien demuestre su solución ganará un premio de ¡un millón de dólares!

Imagination will often carry us to worlds that never were. But without it we go nowhere
Carl Sagan

Conclusión

A pesar de la importancia de las pruebas, en un proyecto de software el tiempo dedicado a las pruebas siempre es menor a lo que debería, y además es lo primero que se trata de recortar cuando el proyecto está atrasado. La Teoría de Gráficas es de enorme ayuda y proporciona herramientas adecuadas para encontrar mejores caminos y trayectorias óptimas de prueba que permiten, si no probar todo (que de hecho es imposible), sí a hacer una prueba estructurada y eficiente.

La historia de este trabajo empezó tratando de explicar la complejidad ciclomática del modelo y el software de McCabe, y fue creciendo al ir encontrando e integrando algunas relaciones entre la Teoría de Gráficas y pruebas específicas de software. Así, terminó siendo lo que es ahora este texto, una relación entre gráficas y pruebas, de ahí el nombre.

Como siempre, hay mucho camino por explorar y se quedan en el tintero múltiples temas, tanto de pruebas como de Teoría de Gráficas, pero el tiempo es finito y quedarán muchas incógnitas de cómo mejorar las pruebas, pero seguramente habrá quienes retomen y puedan mejorar este esfuerzo.

Resumiendo el título de este trabajo con una frase de Beizer, él comenta que probar es sencillo, lo único que un probador necesita hacer es encontrar la gráfica y cubrirla.

Apéndice A. Pequeña explicación de la complejidad ciclomática

Para poder explicar formalmente la definición de complejidad ciclomática, necesitamos utilizar algo de álgebra lineal.

Para cada gráfica *G* con *n* vértices y *e* aristas, se pueden definir varias matrices asociadas; por ejemplo, una de las más usadas es la matriz de adyacencia *A*, de *n* x *n* entradas, donde cada entrada de la matriz se define así:

a_{ij} = 1 si la arista $v_i v_j$ está en la gráfica y 0 si no está.

En el caso de gráficas, la matriz de adyacencia es simétrica sobre la diagonal (ver ejemplo en la figura 27). La matriz de adyacencia nos sirve de ejemplo por varias razones, ya que nos permite enseñar cómo podemos ver una gráfica representada en la memoria de una máquina (como ceros y unos). Al utilizar una gráfica en forma de matriz podemos usar toda la capacidad del álgebra lineal como apoyo en la Teoría de Gráficas.

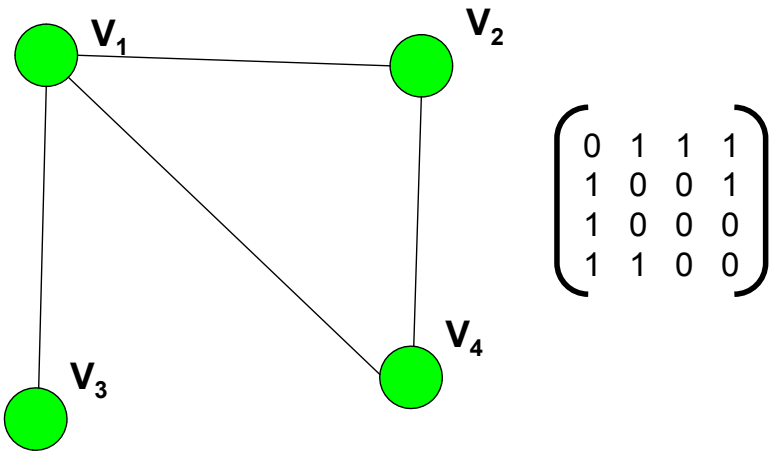


Figura 27. Una gráfica y su matriz de adyacencia.

Esta matriz es usada en muchísimas aplicaciones e incluso en programas de software. Se puede, por ejemplo, definir otra matriz donde en cada entrada se identifica el peso de la arista y, con operaciones de algebra lineal, se obtienen trayectorias óptimas.

También podemos utilizar el álgebra lineal usando espacios vectoriales.

En el libro de Béla Bollobás (1998) se define el espacio de vértices $C_0(G)$ de una gráfica G como el espacio vectorial complejo de todas las funciones de V(G) en el campo de los números complejos. Análogamente, define el espacio de aristas $C_1(G)$ de una gráfica G como el espacio vectorial complejo de todas las funciones de A(G) en los números complejos.

Define después Z(G) como el subespacio de $C_1(G)$ compuesto por todos los ciclos de G (recordemos que un ciclo es un camino donde no se repiten vértices, y en cierta forma, lo podemos ver como un conjunto particular de aristas).

Y él prueba que la dimensión de Z(G) para una gráfica conexa es igual a: $e - n + 1$, que da casualmente nuestro número ciclomático mu.

Es decir, nuestro número ciclomático está directamente relacionado con los ciclos de una digráfica, es formalmente el número de ciclos "básicos" de este espacio vectorial y si vemos a un programa de cómputo como una digráfica, suena lógico usar este número, al ser los ciclos las decisiones que va siguiendo el software.

Si lo vemos de otra manera, podríamos pensar que nuestras trayectorias de prueba son los ciclos dentro de la digráfica y tenemos entonces que encontrar el mínimo número de éstos, ya que los demás serán una combinación lineal de la base de este espacio.

Apéndice B. Una complicación interesante

Ejecutar todas las acciones del modelo está bien, pero una característica muy importante de los modelos y de las pruebas es que muchas veces queremos ponerle "foco" a alguna prueba en especial, por ejemplo, en el caso de probar las transacciones de un cajero puede ser más importante probar los depósitos y retiros que alguna consulta o validación de los mensajes de salida. Un usuario puede saber por experiencia que si dos procesos han corrido seguidos, por ejemplo, un proceso de corte de tarjetas seguido de uno de cálculo de impuestos, uno de ellos falla.

En el ejemplo de la figura 15 ¿que pasaría si queremos además probar específicamente la combinación *ab* o *bg*?, en nuestro ejemplo del software telefónico, querríamos probar el caso de que el ejecutivo contestara y pusiera al cliente en espera (secuencia bg), en la solución podemos ver que la combinación *bg* no está contemplada, esto es debido a que un tour Euleriano nos garantiza pasar por todas las aristas pero no por toda combinación de éstas.

Yellen y Gross (1998) usan un algoritmo conocido como secuencia de De Bruijn, en honor al matemático holandés Nicolaas Govert (Dick) de Bruijn, que nos puede ayudar a hacer esta cobertura. Otro dato curioso de la historia es que las secuencias de De Bruijn se conocen también como secuencias de caja fuerte, ya que describen la secuencia más corta de giros, para probar todas las combinaciones de longitud n y puede ser usada para acortar la búsqueda de un PIN, en lugar de usar un ataque de fuerza bruta.

Para usar esta secuencia de De Bruijn hay que seguir los siguientes pasos:

- Crear una digráfica dual de la original, es decir, una digráfica donde las flechas de la digráfica original sean los vértices de la nueva digráfica.

- Si en la digráfica original, una flecha *1* entra a un vértice y la flecha *2* sale del mismo vértice, se crea una flecha en la digráfica dual del vértice *1* al *2*.

Por ejemplo, si tomamos a la figura 15 como nuestra digráfica original, su digráfica dual quedaría de la siguiente manera:

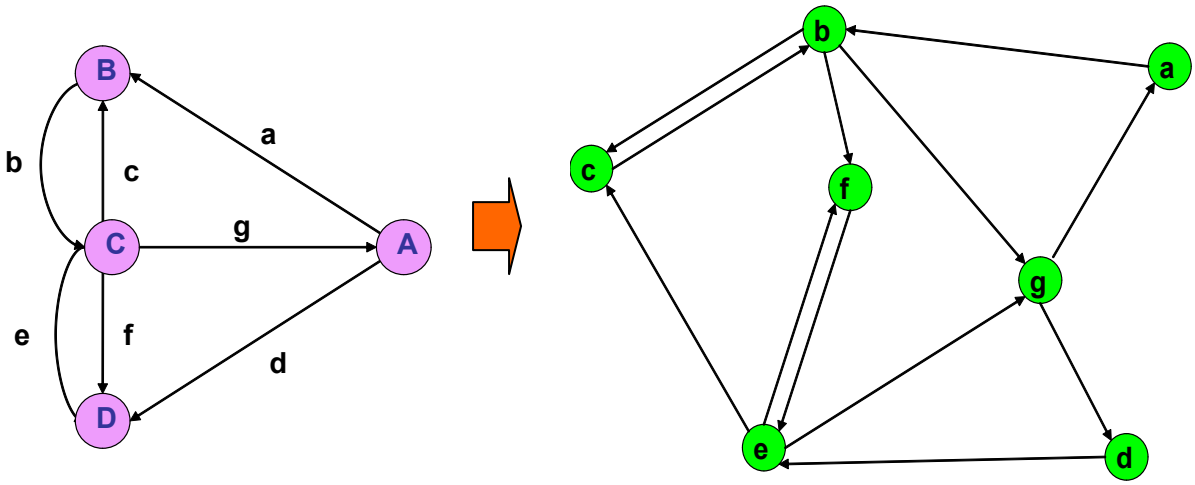


Figura 28. Figura 15 y su digráfica dual

Ahora "Eulerizamos" está digráfica duplicando las flechas que se requieren para poner polaridad 0 a todos los vértices de esta nueva digráfica y buscamos un paseo Euleriano a través de la nueva digráfica (figura 29).

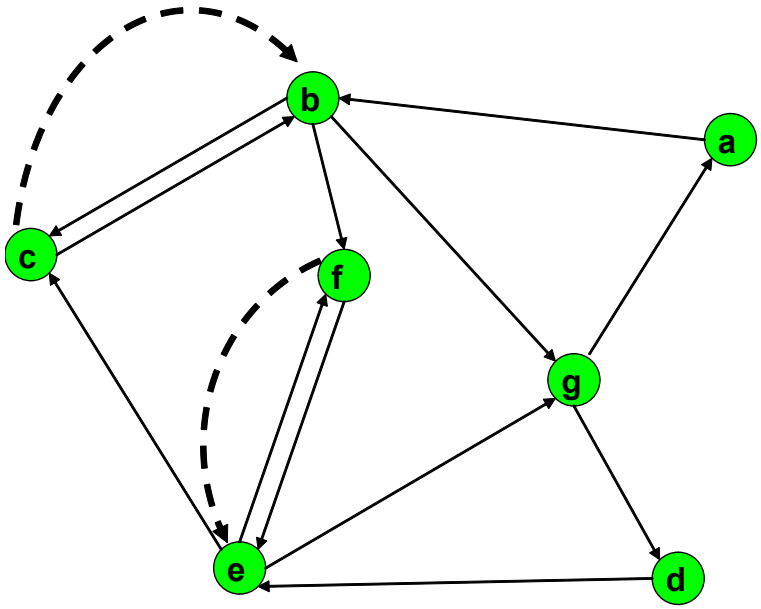


Figura 29. Digráfica dual con aristas virtuales.

Hay que notar que debemos de duplicar flechas existentes para obtener la digráfica con polaridad 0, pues existe la posibilidad de crear otra digráfica con valores de polaridad 0 uniendo vértices con otras flechas que no existen (por ejemplo *ce* y *fb*), que no darían un resultado válido, ya que el camino resultante no existiría en la digráfica original y se obtendría un paseo Euleriano imposible de recorrer.

La secuencia generada a través de un paseo Euleriano de la figura 29 será la combinación de longitud 2 que cubre la gráfica:

c b f e c ***b g*** *d e f e g a b*

Aquí se puede ver que todas las combinaciones de longitud dos ya están en este ciclo, incluida la combinación *bg* que buscábamos.

Apéndice C. Breve línea de tiempo

Para comprender un poco la historia de las pruebas de software y la separación de pruebas con la construcción de software, ésta es una pequeña lista de cómo han evolucionado a lo largo del tiempo.

1958 - Jerry Weinberg es líder del grupo de pruebas del proyecto Mercury (IBM), el primer equipo de pruebas independiente.

1961 - Herbert Leeds y Jerry Weinberg publican *Computer Programming Fundamentals*, el primer libro que describe pruebas de software.

1979 - Glenford Myers publica *The Art of Software Testing*, donde ya se ven teorías como clases de equivalencia, gráficas de causa/efecto, pruebas unitarias.

1979 - McCabe publica su medida de complejidad.

1979 - Winston Royce publica *Managing the development of large software systems*, el primer artículo de administración de software, que tiene un apartado de pruebas.

1982 - En Motorola se crea la metodología *Six Sigma* de mejora de procesos que ya incluye los procesos de prueba.

1989 - Boris Beizer publica *Software Testing Techniques* donde explica que todo programa puede describirse como una gráfica dirigida; así como casos de prueba, el modelo V y otros puntos importantes.

1991 - A partir de este año se crean herramientas de automatización de casos, por ejemplo Winrunner, software para pruebas y herramientas de administración.

1992 - Primer evento dedicado a pruebas de software (*STAR*), actualmente hay eventos en Estados Unidos (este y oeste) y en Europa (Eurostar).

1996 - Jorgensen publica *Software Testing, a Craftman's Approach*.

1997 - Aparecen los primeros talleres o ‘workshops’ (Los Altos Workshop Software Testing) dedicados exclusivamente a pruebas.

1997 - Brian Marick publica *When Should be a Test Automated?*, ya que con el auge de automatización se desvirtúa el proceso de pruebas automáticas.

1999 - Koomen y Pol publican *Test Process Improvement*, una guía que es base para muchas empresas de consultoría de pruebas.

2001 - Manfiesto para el desarrollo de Software - Agile Software Development

2002 - Se publica el libro sobre lecciones aprendidas en el área. *Lessons Learned in Software Testing* (Kaner, Bach, Pettichord)

2002 - En Noviembre se funda el ISTQB (International Software Testing Qualifications Board), una organización registrada en Bélgica con certificaciones en el proceso de pruebas.

2003 - Se crea el Test Maturity Model similar al *CMM* de administración de proyectos, creado por el Instituto de Tecnología de Illinois.

2003 - Lee Copeland publica *Key Test Design Techniques* que unifica algunos temas de pruebas y los empieza a hacer públicos en las conferencias de pruebas.

2008 - 2009 Agile Testing. Dado el auge en la metodología de desarrollo de Agile, la parte de pruebas no puede quedarse atrás y empiezan a publicarse apartados específicos sobre pruebas, entre otros los de Copeland y de Pettichord.

Bibliografía

Amman P. 2008. Offutt Jeff, *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK.

Appel K., Haken W. 1989. *Every Planar Graph is Four Colorable.* Contemporary Mathematics v 98, American Mathematical Society.

Appel K., Haken W. 1977. *The solution of the four color map problem.* Scientific American 237(4): 108-121.

Beizer, B. 1990. *Software Testing Techniques*. 2nd Edition. International Thomson Computer Press.

Blackburn M., Knickerbocker R. 2001. *Mars Polar Lander Fault Identification Using Model-based Testing*. Software Productivity Consortium/T-VEC papers.

Biggs N.L, Lloyd E.K. Wilson R.J. 1976. *Graph Theory* 1736-1936. Clarendon Press, Oxford.

Bodin L., Tucker A. 1983. *A Model for Municipal Street Sweeping Operations*. En: Modules in Applied Mathematics Vol. 3: Discrete and System Models.

Bollobás, Béla 1998. *Modern Graph Theory*. Graduate Texts in Mathematics, vol. 184, Springer, New York.

Bondy J.A., Murty U.S.R. 1976. *Graph Theory with Applications*. Elsevier North-Holland.

Costea A. 2007. *On measuring software complexity*. Journal of Applied Quantitative Methods

Dharwadker A. 2006. *The Vertex Cover Algorithm*
(http://www.dharwadker.org/vertex_cover)

Filiol E., Edouard F., Alessandro G., Moquet B., Roblot G. 2007. *Combinatorial optimisation of worm propagation on an unknown network*. Proceedings of the World Academy of Sciences, Engineering and Technology, 23:

Fleury. 1883. *Deux problemes de geometrie de situation*. Journal de Mathematiques Elementaires 1883, 257-261.

Frank H., Frisch I.T. 1971. *Communication, Transmission, and Transportation Networks*. Addison-Wesley.

Geppert, L. 2004. *Lost Radio Contact Leaves Pilots on Their Own*, IEEE Spectrum, North American Edition

Jorgensen P.C. 2008. *Software Testing a Craftsman's Approach*, Third Edition. Auerbach Publications.

Kasyanov V., Evstigneev N., Vladimir A. 2000. *Graph Theory for Programmers - Algorithms for*. 1st edition. Kluwer Academia Publishers, Springer.

Koomen T., Pol M. 1999. *Test Process Improvement - a practical step by step guide to structured testing*. Addison Wesley.

Kwan M.K. 1962. *Graphic programming using odd or even points*. Chinese Mathematics 1:273-277.

McCabe T.J., Butler C.W. 1989. *Design Complexity Measurement and Testing*, Communications of the ACM, Volume 32 Number 12. December 1989.

McCabe T., McCabe J. *IQ Enterprise Edition* (http://www.mccabe.com/index.htm)

Nyman N. 1998. *GUI application testing with dumb monkeys*. Proceedings of STAR West.

Pan J. 1999. *Software Reliability*. Carnegie Mellon University 18-849b Dependable Embedded Systems.

Rajappa V., Biradar A., Panda S. 2008. *Efficient Software Test Case Generation Using Genetic Algorithm Based Graph Theory*. Emerging Trends in Engineering and Technology, First International Conference on 16-18 July 2008, pp. 298-303.

Traveling Salesman Problem, Georgia Tech, http://www.tsp.gatech.edu/

Thibodeau P. 2002. *Study*: *Buggy software costs users, vendors nearly $60B annually* Computerworld, June 25.

Volkmann, L. "Estimations for the Number of Cycles in a Graph." Per. Math. Hungar. 33, 153-161, 1996.

Watson A.H., McCabe T.J. 1996. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. Computer Systems Laboratory, National Institute of Standards and Technology (NIST), September issue.

Yellen J., Gross J.L. 1998. *Graph Theory and Its Applications*. 1st edition. CRC Press.