Guía de Teoría de grafos

Autor: Wilmer Bandres

Tabla de Contenidos

1 Introducción a Grafos **2**

1.1 Ejemplo de grafos 2

1.2 Definición formal de un grafo 3

1.3 Algunos tipos de grafos 5

1.4 Coloreo de un grafo 7

2 Representación de grafos **9**

2.1 Representación de grafos mediante lista de adyacencias 9

2.2 Representación de grafos mediante una matriz 10

2.3 Representación implicita de grafos 10

2.4 Representación de un problema como un grafo 11

3 Conectividad en grafos **13**

3.1 Estructura de un grafo 13

3.2 Algebra de caminos 14

3.2.1 Propiedades de los caminos de un grafo 15

3.2.2 Más tipos de grafos 16

3.3 Alcanzabilidad 20

3.3.1 Matriz de alcanzabilidad 20

3.3.2 Algoritmo de Roy-Warshall 21

3.3.3 Clausura transitiva de un grafo 23

3.4 Componentes conexas 23

3.4.1 Calculo de componentes fuertemente conexas 24

3.5 Conjuntos de articulación 25

4 Recorrido de grafos **28**

4.1 El recorrrido de grafos 28

4.2 Búsqueda en profundidad o Depth First Search (DFS) 29

4.2.1 Implementación iterativa de un DFS 31

4.3 Búsqueda en amplitud o Breadth First Search (BFS) 32

4.4 Ordenamiento topológico 35

4.5 Algoritmo de Tarjan para componentes fuertemente conexas 37

4.6 2-SAT 40

5 Caminos más cortos: caminos de costo mínimo **42**

5.1 Introducción a caminos de costo mínimo 42

5.2 Camino de costo mínimo en un DAG 45

5.3 Dijkstra 45

5.4 Bellmand-Ford: caminos cortos en grafos con arcos costo negativo . 47

5.5 Caminos simples de costo máximo y mínimo 49
5.6 Caminos de costo mínimo entre cada par de vertices 49

6 Búsqueda informada **52**
6.1 Algoritmo A^* 53
6.2 Admisibilidad de A^* 54

7 Árboles **57**
7.1 Arboles y arborescencias 57
7.2 Arbol cobertor de costo mínimo (Minimum Spanning Tree) 58
7.2.1 Algoritmo de Kruskal 59
7.2.2 Algoritmo de Prim 60
7.3 Diámetro de un árbol 61

8 Algoritmos de flujo máximo **62**
8.1 Flujos y flujo máximo 62
8.2 Algoritmo de Ford-Fulkerson 64
8.2.1 Capacidades residuales 64
8.2.2 Cortes en una red de flujo 66

9 Miscellaneous **69**
9.1 Rule 69

10 Some Common Errors **70**

11 Further Reading **71**
11.1 Websites and Tutorials 71
11.2 Introduction/Commands/and Overall Good References 71
11.2.1 Writing First Document 71
11.2.2 Brief Introduction/List of Commands 71
11.2.3 In-Depth LaTeX Guide 72
11.3 Boxes 72
11.4 Typesetting Math Documents 72
11.4.1 Creating Math Documents 72
11.4.2 Symbols 72
11.4.3 Typing Math Functions/Equations 73
11.5 Graphics 73
11.5.1 Images 73
11.5.2 Colors 73

25 References **75**

About This File

This file was created for the benefit of all teachers and students wanting to use Latex for tests/exams/lessons/thesis/articles etc.
The entirety of the contents within this file, and folder, are free for public use.

1

SECCIÓN

Introducción a Grafos

1.1 Ejemplo de grafos

Empecemos dando ejemplos de grafos para tener una idea de lo que son

(a) Un nodo

(b) Dos nodos

(c) Dos nodos unidos por una arista

(d) Grafo dirigido

(e) Multigrafo

Los grafos estan compuesto por nodos y aristas. Las aristas conectan nodos, y pueden haber multiples aristas conectando un par de nodos. Los grafos representan una de las estructuras

mas genéricas en computación y tienen gran utilidad en la vida diaria, por ejemplo Google Maps: cada ciudad puede ser representada por un nodo y las calles o vías conectando un par de ciudades serían las aristas conectando un par de nodos en el grafo.

1.2 Definición formal de un grafo

Pasemos ahora a dar una definición formal de un grafo. Anteriorment dijimos que un grafo esta formado por un conjunto de nodos y un conjunto de aristas, entonces:

Definición 1.2.1: Grafo

Un grafo, o grafo simple, es un par $G = (V, E)$, donde V es el conjunto de vertices de G y E es el conjunto de aristas de ese grafo. E es un conjunto de pares de vertices, cada lado puede ser escrito como $e = \{x, y\} \in E$ donde $x, y \in V$.

Nótese que cada vertice del grafo está identificado por alguna letra o etiqueta, y los lados o aristas también. La diferencia radica en que las aristas están representadas como un conjunto de exactamente dos vertices significando que si dos vertices $x, y \in V$ estan unidos por una arista $e = \{x, y\}$, entonces es posible ir desde x hasta y y viceversa, ya que es un grafo no dirigido.
Decimos que es un grafo simple o grafo ya que bajo esta definición estamos asumiendo que dos nodos están unidos por a lo sumo una arista. Además asumimos que el grafo no tiene bucles: lados de la forma $\{x, x\}$.
Con esta definición es posible representar todo tipo de grafos simples, por ejemplo el grafo vacío (aquel que no tiene aristas o vertices) es simplemente $G = (\emptyset, \emptyset)$. Un grafo con 3 vertices con todos los posible lados del mismo sería por ejemplo $G = (\{x, y, z\}, \{\{x, y\}, \{x, z\}, \{y, z\}\})$, que tambien puede ser representado como $G = (\{x, y, z\}, \{e_1, e_2, e_3\})$ con $e_1 = \{x, y\}$, $e_2 = \{y, z\}$ y $e_3 = \{x, z\}$.
Ahora bien, también necesitamos formalizar nuestra definición de grafo con lados que permiten ir en una sola dirección.

Definición 1.2.2: Grafo dirigido

Un grafo dirigido es un par $G = (V, E)$, donde V es el conjunto de vertices de G y $E \subseteq V \times V$ es el conjunto de aristas de ese grafo. E es un conjunto de pares **ordenados** de vertices y cada lado puede ser escrito como $e = (x, y) \in E$ donde $x, y \in V$.

La diferencia entre un grafo dirigido un un grafo no dirigido (o grafo) radica en como definimos sus aristas. En un grafo dirigido $e_1 = (x, y) \neq (y, x) = e_2$, (x, y) indica que se puede ir desde x hasta y mediante e_1, pero no al revés; y (y, x) indica que se puede ir desde y hasta x mediante e_2, pero no al revés. Por otro lado, en un grafo no dirigido $\{x, y\} = \{y, x\}$, fíjese que por esto justamente usamos conjuntos para representar las aristas no dirigidas, ya que en teoría de conjuntos $\{x, y, z, w\} = \{x, z, y, w\} = \{z, w, x, y\}$. Además en un grafo no dirigido, el lado $\{x, y\}$ permite ir desde x hasta y, y viceversa.

Definición 1.2.3: Grafo completo

Un grafo **completo** es un grafo en el que cada par de vertices están unidos por una arista.

Esto quiere decir que un grafo completo puede ser escrito como $G = (V, (V \times V) \setminus \{(x, x) : \forall x \in V\})$.

Ejemplos de este tipo de grafos puede ser el grafo vacío; un grafo con dos nodos y una arista $G = (\{x, y\}, \{\{x, y\}\})$; un grafo en forma de un triángulo $G = (\{x, y, z\}, \{\{x, y\}, \{x, z\}, \{y, z\}\})$.

Definición 1.2.4: Grafo inducido

Un grafo **inducido** $G[V']$ de un grafo $G = (V, E)$, es un grafo formado por un subconjunto $V' \subseteq V$ de vertices de G y todas las aristas del grafo original G que conectan a un par de vertices de V'.

Supongamos que tenemos un grafo $G = (V = \{x, y, z, w\}, E = \{\{x, y\}, \{z, w\}, \{y, z\}\})$, el grafo inducido por $V' = \{x, y, z\}$ es el grafo $G' = (\{x, y, z\}, \{\{x, y\}, \{y, z\}\})$. El grafo inducido por $V'' = \{x, y\}$ es $G'' = (\{x, y\}, \{\{x, y\}\})$. El grafo inducido por $\emptyset$ es el grafo vacío $G = (\emptyset, \emptyset)$.

Definición 1.2.5: Clique de un grafo

Un **clique** de un grafo $G = (V, E)$ es un subconjunto de vertices $V' \subseteq V$ de un grafo cuyo grafo inducido es un grafo completo, es decir $G[V'] = (V', V' \times V')$.

Lo siguiente que podríamos definir es un clique maximo y uno maximal, con el objetivo de empezar a buscar propiedades de los grafos.

Definición 1.2.6: Clique maximal de un grafo

Un **clique maximal** es un clique V' de un grafo $G = (V, E)$ en el que si añadieramos un vertice cuaquiera $x \in V$, entonces $V' \bigcup \{x\}$ no es un clique.

Definición 1.2.7: Clique maximo de un grafo

Un **clique maximo** es un clique $V' \subseteq V$ de un grafo $G = (V, E)$ cuyo tamaño entre todos los cliques de G, es maximo.

El clique maximo de un grafo es un problema NP-completo, esto quiere decir que posiblemente no exista una solucion "eficiente" que pueda resolver el problema. Próximamente veremos algunas propiedades del mismo pero primero procedamos a agregar mas definiciones a nuestro repertorio...

Definición 1.2.8: Lados adyacentes a un nodo

Dado un grafo $G = (V, E)$, decimos que un lado $e \in E$ es adyacente a un nodo $x \in V$ si $x \in e$

Definición 1.2.9: Nodos incidentes a una arista

Dado un grafo $G = (V, E)$, decimos que un nodo $x \in V$ es incidente a la arista $e \in E$ si e es adyacente a x

Definición 1.2.10: Nodos adyacentes

Dado un grafo $G = (V, E)$, decimos que los nodos $x, y \in V$ son adyacentes si existe una arista $e \in E$ tal que $e = \{x, y\}$

Definición 1.2.11: Grado de un nodo

El grado de un nodo es $n \in V$ es número de lados adyacentes al nodo en el mismo grafo lo denotamos como $|\delta(n)|$. Donde $\delta(n)$ es el conjunto de arcos adyacentes a n.

Definición 1.2.12: Grado interior de un nodo (Grafos dirigidos)

El grado interior de un nodo $n \in V$ en un grafo dirigido $G = (V, E)$ es número de lados adyacentes a n que contienen a n como el nodo de llegada. Lo denotamos $|\delta^-(n)|$

Definición 1.2.13: Grado exterior de un nodo (Grafos dirigidos)

El grado exterior de un nodo $n \in V$ en un grafo dirigido $G = (V, E)$ es número de lados adyacentes a n que contienen a n como el nodo de salida. Lo denotamos $|\delta^+(n)|$

Supongamos que tenemos el grafo $G = (\{x, y, z\}, \{\{x, y\}, \{x, z\}\})$, el grado de x es $|\delta(x)| = 2$, y los grados de y y z son $\delta(y) = \delta(z) = 1$. Y ahora con esta definición llegamos a nuestro primer teorema :)

Teorema 1.2.14: *La suma de los grados de los vertices de un grafo $G = (V, E)$ es dos veces el numero de lados del mismo*

: *Para esto veamos que valor tiene $\sum_{x \in V} |\delta(x)|$, en esta suma cada lado suma una unidad a el grado de cada uno de los vertices que están conectados por ese lado, esto quiere decir que* $\sum_{x \in V} |\delta(x)| = \sum_{x \in V} \sum_{\substack{e \in E \\ x \in e}} 1 = \sum_{e \in E} \sum_{\substack{x \in V \\ x \in e}} 1 = 2 \times |E| \blacksquare$

1.3 Algunos tipos de grafos

Por ahora hemos estado prohibiendo tener más de un solo arco conectando un par de nodos. Sin embargo, es posible extender la definición de grafo de la siguiente manera

Definición 1.3.1: Multigrafo

Un multigrafo, es un par $G = (V, E)$, donde V es el conjunto de vertices de G y E es un **multiconjunto** de aristas de ese grafo. E es un multiconjunto de pares de vertices, cada lado puede ser escrito como $e = \{x, y\} \in E$ donde $x, y \in V$

Bajo la definición de multigrafo, es posible tener entonces más deuna arista entre un par de vertices. A partir de este punto lo más seguro es que solo trabajemos con grafos simples a menos que se diga lo contrario.

Por otro lado, hay otros tipo de grafos simples que son de mucho interés por su uso práctico:

Definición 1.3.2: Grafo bipartito

Un grafo bipartito, es un par $G = (V \dot\cup V', E)$, donde V, V' son conjuntos de vertices **disjuntos** de G y E es un conjunto de aristas de ese grafo. E es un conjunto de pares de vertices donde cada lado puede ser escrito como $e = \{x, y\} \in E$ con $x \in V$ e $y \in V'$, o viceversa

El símbolo $\dot\cup$ simplemente representa la union disjunta de conjuntos.

Definición 1.3.3: Grafo regular

Un grafo regular, es un grafo $G = (V, E)$ en el que cada nodo $x \in V$ cumple con $\delta(x) = k$ para alguna constante k. Un grafo regular en el que cada nodo tiene grado k es llamado un grafo $k-$regular. En particular, en un grafo $2-$regular cada nodo tiene dos lados adyacentes.

Fíjense que bajo esta definición el grafo vacío es un grafo $k-$regular para cualquier k. Sin embargo, vamos a decir para evitarnos problemas que el grafo vacío es un grafo $0-$regular
Ahora imaginemos que queremos intentar dibujar un grafo, y quisieramos saber si es posible dibujar ese grafo sin que dos aristas se intercepten en su dibujo. Esto no siempre esposible, y de hecho es un problema muy importante en diseño de circuitos donde se intenta unir puntos de contactos de un circuito sin que se crucen cables por donde pasa la electricidad.

Definición 1.3.4: Grafo planar

Un grafo es planar si puede ser dibujado en un plano de cualquier manera tal que dos aristas no se interceptan.

Un grafo completo K_5 con 5 nodos no es planar, además es el grafo mínimo (con menor cantidad de nodos) posible que no es plano.

Definición 1.3.5: K_n

K_n es el grafo completo de n nodos

Definición 1.3.6: $K_{m,n}$

$K_{m,n}$ es el grafo completo **bipartito** de $m + n$ nodos, donde los nodos se dividen en dos conjuntos disjuntos de V_1, V_2 con $|V_1| = m$ y $|V_2| = n$ y cada uno de los nodos de V_1 esta conectado con cada nodo de V_2 mediante una arista.

El grafo $K_{3,3}$ no es un grafo planar. Además para cada $n > 3$ el grafo $K_{n,n}$ no es planar ya que entre otras cosas incluye al grafo $K_{3,3}$.

Definición 1.3.7: Grafo conexo

Un grafo $G = (V, E)$ es conexo si es imposible dividir V en dos conjuntos disjuntos $V_1 \dot{\cup} V_2 = V$ tal que no existe un arco $e = \{x, y\} \in E$ con $x \in V_1$ e $y \in V_2$

Definición 1.3.8: Grafo disconexo

Es un grafo que no es conexo

Existe una característica de los grafos planos a la que se le llama formula de Euler. Esta formula dice lo siguiente

Teorema 1.3.9 (Fórmula de Euler): *Para un grafo conexo plano $G = (V, E)$ que contiene al menos un arco, si r es el número de regiones creadas al dibujar G entonces $r = |E| - |V| + 2$*

: *Procedemos por inducción en $|E|$. En el caso de $|E| = 1$ solo existe un arco entre un par de nodos un arco y esos dos nodos son los únicos nodos del grafo ya que es conexo. Además existe una sola region en el plano ya que ese lado solo no puede formar una region cerrada, por lo $tanto$ $1 = 1-2+2$. Ahora asumimos que el teorema se cumple para $|E| = m-1$ aristas y probamos para m aristas.*
Ahora digamos que tenemos un grafo plano conexo de m arcos y removemos un arco y el grafo sigue siendo conexo. Tenemos un grafo cualquiera conexo y plano con $m-1$ lados. Existen dos casos entonces al agregar el arco que acabamos de remover:

- *El arco es agregado a un nodo nuevo que estaba aislado hasta ahora. En ese caso entonces el número de nodos aumenta en 1, y el número de arcos también. Además el agregar ese arco no parte ninguna región en dos y agregar una nueva ya que el grafo es plano. Esto quiere decir que la nueva fórmula no cambia ya que insertando los nuevos valores de $|V|$ y E, queda en $r = |E|+1-(|V|+1)+2$ que es cierto por inducción.*

- *El arco es agregado entre dos nodos que ya estaban en el grafo plano de $m-1$ lados. En ese caso agregar el arco cierra o crea una nueva region por lo tanto $|E|$ y r aumentan en 1 y el teorema se cumple por inducci"on.* ■

Definición 1.3.10: Grafo estrella

Es un grafo con n nodos, es un grafo conexo en donde $n-1$ vertices tienen grado 1 y 1 nodo tiene grafo $n-1$

Otra forma de definir el grafo estrella de n nodos es simplemente el grafo $K_{n-1,1}$.

1.4 Coloreo de un grafo

Dado un grafo, imaginemos que podemos pintar con algun color donde podemos colorear dos nodos distintos con el mismo color. Si pudieramos colorear los nodos del grafo de cualquier manera entonces no existiría problema alguno, pero hagamos supongamos que queremos colorear el grafo con la restricción de que si dos nodos estan unidos por un arco entonces tienen que tener diferentes colores.

Decimos que un grafo es $k-$coloreable, si es posible colorear el grafo con a lo sumo k colores.

Definición 1.4.1: Número cromático de un grafo

El número cromático $\chi(G)$ de un grafo G, es el mínimo número de colores con el que se puede colorear G sin que dos nodos adyacentes tengan el mismo color

Lema 1: Número cromático de un grafo bipartito

Para un grafo bipartito G con al menos un lado $\chi(G) = 2$

: *Dado que un grafo bipartito puede ser dividido en un grafo cuyos dos conjuntos disjuntos A y B nombremos a dos colores, el color 1 va a ser asignado a cada nodo de A y el color 2 a cada nodo de B, como ningun nodo en A esta conectado con un nodo de A mismo y lo mismo para B, el coloreo sugerido es un coloreo valido de maximo dos colores* ■

SECCIÓN 2

Representación de grafos

Un grafo puede estar representado de manera gráfica, es decir mediante dibujos y es una manera válida de representarlos que ayuda a su visualización y quizás a descubrir propiedades del mismo; también puede estar representado mediante relaciones (teoría de conjuntos) en donde dos elementos están relacionados si y solo si existe un arco entre esos dos elementos (nodos); por último un grafo puede estar representado en memoria de muchas maneras. La estructura de datos utilizada para ello afecta entonces de manera directa cualquier manipulación sobre el grafo: agregar un nodo, agregar un arco, remover un nodo, remover un arco, etc. De la misma manera que la forma de representar un arreglo afecta el tiempo de ejecución de los algoritmos de ordenamiento, una "mala" (ineficiente) representación del grafo afectará los algoritmos que vamos a ver en un futuro.

2.1 Representación de grafos mediante lista de adyacencias

Si queremos representar un grafo $G = (V, E)$ una manera de representarlos es mediante listas de adyacencias, donde para cada nodo $a \in V$ guardamos una lista $ady(a)$ de aquellos nodos adyacentes a a. Bajo esta representación es necesario entonces guardar V listas de adyacencias, donde cada una puede variar de tamaño.
Además es necesario saber donde se guardan las listas, puede ser en un arreglo de listas o en una listas de listas, dependiendo del tipo de representación buscar si dos nodos están conectados tiene distintos tiempos de ejecución.

Si tenemos una lista de listas, y queremos saber si dos nodos $a, b \in V$ están unidos mediante una arista debemos buscar primero por ejemplo la lista de adyacencia de a y luego en esa lista

buscar a b. Si asumimos que el grafo es simple, esto quiere decir que buscar si dos nodos estan conectados tomaría un tiempo igual a $2 \cdot |V|$. Si en lugar de tener listas de listas tuvieramos un arreglo de listas entonces el tiempo total está acotado por $|V|$ simlpemente.
Sin embargo, una ventaja de la representación de listas de listas es que es posible agregar un nodo de manera muy sencilla ya que es simplemente crear una nueva lista vacía (representando la lista de adyacencia del nodo nuevo) y agregandola a la lista de listas en tiempo constante mediante el uso del apuntador al último nodo creado hasta el momento. Por otro lado, agregar un nodo a la representación de arreglo de listas puede requerir relocalizar todo el arreglo en el caso de que el arreglo esté saturado.

2.2 Representación de grafos mediante una matriz

La representación matricial de un grafo $G = (V, E)$ con $|V| = n$ nodos requiere una matriz bidimensional binaria A (cada entrada es 0 o 1), de tamaño $n \times n$ en donde $A_{a,b} = 1 \iff \{a, b\} \in E$. Al ser una representación matricial, tiene la ventaja de que buscar si dos nodos son adyacentes tomaría tiempo constante. Sin embargo, agregar o quitar un nodo requeriría relocalizar la matriz completamente.

Una desventaja bajo la representación matricial es que si un grafo no tiene demasiadas aristas, esto quiere decir que muchas entradas de A van a ser 0 y siempre necesitaríamos tener en memoria $|V|^2$ entradas de la matriz. Note que esta representación es independiente del numero de lados del grafo.

2.3 Representación implicita de grafos

Ahora supogamos que cada nodo de un grafo $G = (V, E)$ tiene un número asociado. Podríamos definir una función binaria $f : V \times V \Rightarrow \{0, 1\}$ sobre los pares de nodos del grafo en donde si $a, b \in V$ entonces $\{a, b\} \in E \iff f(a, b) = 1$. Bajo esta representación es posible representar el grafo en memoria solo con un arreglo de $|V|$ entradas (con los números asociados) aunque el grafo tenga una cantidad de lados muy grande.

Un ejemplo de este tipo de grafos podría ser imaginar un grafo completo con n nodos donde $f(a, b) = 1 \forall a, b \in V$, en este caso nos ahorraríamos representar el grafo y sus $|E| = |V|^2$ aristas con solo usar una cantidad de memoria $O(|V|)$.
En general, el tipo de representación óptima en memoria que queramos usar depende del tipo de problema al que nos estemos enfrentando. Si sabemos que un grafo tiene siempre la misma cantidad de nodos y nuestra operación a optimizar es la de agregar y quitar lados, entonces la representación matricial parece ser la mejor, ya que en tiempo constante podríamos hacer flip del los bits que representa el lado que estamos agregando / quitando. Fíjese que digo bits y no bit, porque en la representación matricial de un grafo simple no dirigido, el lado $\{a, b\}$ puede ser chequeado al ver cualquiera de los bits $A_{a,b}$ o $A_{b,a}$ ya que la matriz de adyacencia en un grafo no dirigido es una matriz simétrica.

2.4 Representación de un problema como un grafo

Los grafos son unas de las estructuras más genéricas que existen en computaicón y con ellos podemos representar una gran variedad de problemas y esto nos genera la ventaja de poder resolver con un mismo algoritmo problemas que quizás no estan relacionados pero que en su representación lucen similares. Obviamente dependiendo del problema y de su semántica quizás en una primera instancia no se parecen a otros hasta que buscamos representarlos de alguna otra manera.
Ahora enfoquemonos en un solo problema: el cubo de Rubik

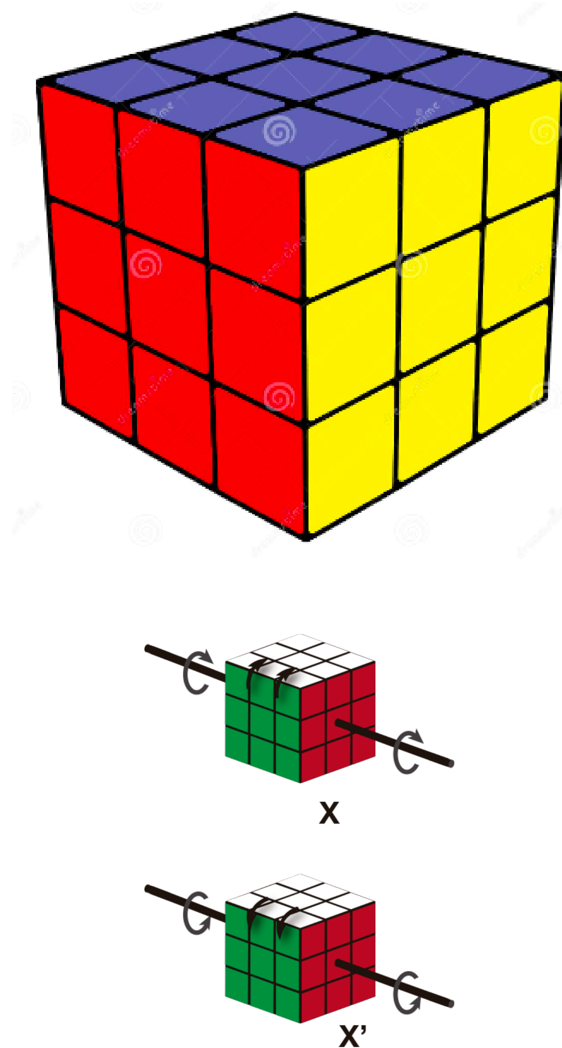


Figure 2: Cubo de Rubik y sus movimientos

En este problema la idea es intentar mover el cubo hasta poder dejar todos los cuadritos del mismo color en una misma cara, para cada color. Ahora bien, lo que queremos es ordenar el cubo. Ahora imaginemos que tenemos una configuración inicial cualquiera del cubo, nuestro problema u objetivo es poder llegar a la configuración donde todos los colores estan en la misma cara, para cada color. Por otro lado, la solución al problema no es solo resolverlo sino hacerlo lo más rápido posible. Entonces dada una configuración inicial del cubo (nuestro nodo inicial)

queremos encontrar el camino mas corto hasta dejarlo ordenado en donde un paso es simplemente un giro en alguna dirección de una fila o columna del cubo.

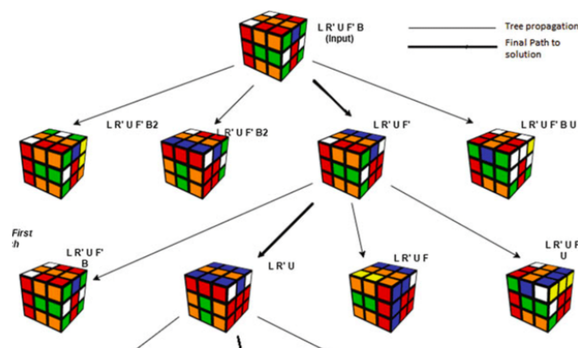


Figure 3: Grafo para representar el problema del cubo de Rubik

En la figura 3 podemos ver entonces una manera de representar el problema del cubo de Rubik, y también podemos ver que en este caso fue representado como un grafo dirigido, sin embargo es posible de representarlo como un grafo no dirigido. Luego, tomando en cuenta que cada paso o giro tiene costo 1 (en tiempo o en alguna medida que quisieramos utilizar) nuestro objetivo sería encontrar el camino mas corto desde la configuración inicial hasta la ordenada. El problema de encontrar el camino mas corto es entonces un problema general de un grafo, no es específico para el cubo de Rubik y es algo que aprenderemos a resolver pronto.

Un problema puede ser representado de varias maneras, se posible también armar un grafo con paso de tamaño 2 por ejemplo, en donde cada arco representan dos giros exactamente. El problema de esta representación es que si del nodo inicial al final el camino mas corto tiene un costo (o cantidad de pasos) que es impar entonces no es posible encontrarlo. Entonces, es una necesidad poder representar el problema de tal manera que tenga sentido y también poder decir que al encontrar una solución, esa solución corresponde a algo válido en el mundo real. En el caso del cubo de Rubik existen configuraciones que no son alcanzables entonces es importante que nuestro problema de grafos no caiga en ellas.

Representar un problema como un grafo es literalmente un arte y no existe método fijo a seguir para hacerlo de manera "correcta", es decir, que es algo que se aprende con la práctica y espero que este curso les ayude.

SECCIÓN 3

Conectividad en grafos

En cualquier representación de los grafos es muy natural preguntarse si dos nodos están conectados de alguna manera, si alguien parado en un nodo puede alcanzar un segundo nodo, si un nodo está aislado, de cuantas formas se puede llegar de un nodo a otro, etc. Todos estas preguntas y más tienen que ver con cómo un grafo esta "construido", es decir, dependen de su estructura.

3.1 Estructura de un grafo

Un grafo puede tener muchas definiciones o propiedades asociadas, empecemos entonces por definir las más básicas

Definición 3.1.1: Camino Simple

Dado un grafo $G = (V, E)$, un camino simple, es una secuencia alternante $(n_1, e_1, n_2, e_2, ..., e_k, n_{k+1})$ de nodos y lados, que empieza en un nodo y termina en otro nodo, con la propiedad de que $n_1, ..., n_{k+1} \in V$ y $e_1, ..., e_k \in E$ y además no hay ningún nodo repetido

Si algún nodo se repite entonces diremos que el camino no es un camino simple, es simplemente un **camino**. Además llamamos **camino elemental** al camino simple que contiene un solo nodo.

Definición 3.1.2: Longitud de un camino

La longitud de un camino es simplemente la cantidad de arcos (incluyendo repeticiones en el caso de caminos no simples) que estan contenidos en el camino

Existen un tipo de camino que nos podrían interesar identificar en las estructuras de un grafo,

esto son los ciclos.

Definición 3.1.3: Ciclo (o ciclo simple)

Un ciclo es simplemente un camino que empieza y termina en el mismo nodo y además ningún nodo se repite exceptuando el primero.

Si un ciclo tiene k nodos distintos, entonces existen k formas distintas de representar el mismo ciclo ya que cada nodo puede ser el principio del camino de ese ciclo. Es importante notar que los conceptos representados hasta ahora son independientes del tipo de grafo ya que éste puede ser orientado (dirigido) o no y aún así la definiciones siguen siendo válidas.

Definición 3.1.4: Circuito (o ciclo simple)

Un circuito es simplemente la definición de ciclo para grafos dirigidos

3.2 Algebra de caminos

Es posible definir un algebra con los caminos de un grafo. Para definir un algebra entonces es necesario definir operaciones entre los elementos, que en este caso son los caminos.

- Longitud de un camino: $l : C(G) \to \mathbb{N}$, donde $C(G)$ es el conjunto de caminos posibles de un grafo G, y dado un camino retorna su longitud.

- Concatenación de caminos: $cc : C(G) \times C(G) \to C(G)$, que es una operación que concatena dos caminos si el final de uno es el comienzo de otro, es decir si en un grafo $G = (V, E)$ existen dos caminos $c1 = (n_1, e_1, ..., e_k, n_{k+1}), c_2 = (n'_1, e'_1, ..., e'_k, n'_{k+1}) \in E$, entonces podemos concatenarlos y usar la operación cc sólo si $n_{k+1} = n'_1$. En ese caso entonces la cadena resultante sería $(n_1, e_1, ..., e_k, n_{k+1}, e'_1, ..., e'_k, n'_{k+1})$.
  Esta operación no esta siempre definida, la propiedad mencionada anteriormente no se cumple, entonces decimos que el resultado es un camino indefinido que denotaremos ∞. Por otro lado, una propiedad interesante al concatenar dos cadenas c_1, c_2 es que $l(cc(c_1, c_2)) = l(c_1) + l(c_2)$.

- Camino inverso: $inv : C(G) \to C(G)$, si $c = (n_1, e_1, ..., e_k, n_{k+1}) \in C(G)$ entonces $inv(c) = (n_{k+1}, e_k, ..., e_1, n_1)$.

- Agregar nodos a un camino: $Ag : C(G) \times V \to C(G)$ es simplemente la operación de agregar un nodo del grafo a un camino. Esta operacion es válida si el ultimo nodo del camino a considerar esta conectado con el nodo a agregar en el camino. Es decir, dado $c = (n_1, e_1, ..., e_k, n_{k+1}) \in C(G)$ y n_{k+2}, la operación $Ag(c, n_{k+2})$ solo está definidad si $\{n_{k+1}, n_{k+2}\} \in E$.

- Distancia entre dos nodos: $d : V \times V \to \mathbb{N}$, es una operación que dado $a, b \in V$ entonces $d(a, b)$ es un entero que es la longitud del camino pas corto entre a y b o infinito en el caso de que una cadena entre ellos es inexistente en el grafo a analizar.

Con el algebra ahora definida con ciertas operaciones, entonces es posible ahora empezar a demostrar "verdades" a las que llamamos teoremas, lemas o propiedades (dependiendo de su importancia y puede ser tambien dependiendo de qué tan directo es derivado de las operaciones basicas).

3.2.1 Propiedades de los caminos de un grafo

- Si existe un camino C entre dos nodos $n_1, n_{k+1} \in V$ de $G = (V, E)$, entonces existe un camino simple entre ellos

(Prueba constructiva): *Si existe un camino* $c = (n_1, e_1, ..., e_k, n_{k+1}) \in C(G)$*, y no es un camino simple, entonces en algun punto (si leemos de izquierda a derecha el camino, o si leemos desde el comienzo hasta el final) un nodo se repetirá, supongamos que los indices donde ocurren por primera y segunda vez ese mismo nodo son* i *y* j*, entonces podemos quitar del camino* c *todos los arcos y nodos entre el nodo* i *y su siguiente ocurrencia en el indice* j*, exceptuando la ocurrencia en* i*. El camino resultante es un camino válido desde* n_1 *hasta* n_{k+1} *pero su longitud es menor . Si el camino sigue siendo no simple, entonces repetimos este proceso hasta no encontrar un nodo que se repida y finalmente terminariamos teniendo un camino simple.* ■

(Prueba por inducción): *Supongamos ahora que queremos analizar la siguiente cantidad* $r(C) = \sum_{v \in C} \pi(v)$*, donde* $\pi(v)$ *es la cantidad de repeticiones del nodo* v *en el camino* C*, entonces si* $r(C) = 0$ *es porque* C *es elemental.*

Supondamos que la proposición es cierta para todos los caminos con $r(C) \leq p$ *e intentemos demostrar para* $p+1$*. Como* $r(C) = p+1$ *entonces tenemos que* C *tiene un vertice que se repite. Tomamos dos ocurrencias que ese vertice tiene en C y eliminamos una de ellas con el argumento de la prueba anterior. Cuando esto ocurre* $r(C) \leq p$ *y por inducción existe un camino elemental entre* n_1, n_{k+1} ■

- Dado un grafo $G = (V, E)$, si el grado de cada nodo es mayor o igual a 2, entonces G tiene un ciclo

: *Empecemos un camino simple desde un nodo cualquiera* $n_1 \in V$ *y digamos que nos movemos aleatoriamente a otros nodos, sin importar a cual nos movemos. Asumamos que nos movemos lo más posible y recordemos que no repetimos nodos, ya que es un camino simple. Digamos que el camino que realizamos es* $n_1, e_1, ..., e_k, n_{k+1}$*, como es un camino simple entonces* n_{k+1} *no ha ocurrido antes en el camino. Además sabemos que el grado de* n_{k+1} *cumple que* $deg(n_{k+1}) \geq 2$*, esto quiere decir que* n_{k+1} *tiene aún un arco disponible que sale de el y conecta con alguno de los nodos anteriores (de lo contrario no nos hemos movido lo mas posible que fue lo que dijimos), entonces ese arco existe y crea un ciclo que existe en G* ■

- Dado un grafo $G = (V, E)$ si $|E| \geq |V|$ entonces G contiene un ciclo.

  : *Digamos que el grafo que nos dan es conexo, sino aplicamos el siguiente argumento en cada parte conectada del grafo para encontrar un ciclo en por lo menos una de ellas.*
  Agarramos un nodo cualquiera del mismo, y vamos a empezar a construir un árbol. Empezamos por el nodo inicial y creamos un camino. El grafo resultante hasta ahora es un camino. Luego buscamos un nodo cualquiera que no hayamos agregado al grafo y que se pueda unir a el mismo mediante una arista y lo agregamos. Iteramos sobre este proceso y el grafo resultante es un árbol.
  En cada componente conectada que creamos, creamos un árbol cuya cantidad de arcos es la cantidad de nodos de ese grafo menos 1. Esto quiere decir que hay por lo menos en una de esas componentes un arco que unen dos nodos del mismo pero que no ha sido agregado por el proceso anterior. Debido a que $|E| \geq |V|$, ese arco debe existir. Lo que resulta de aunar ese arco al grafo es un ciclo ■

- Dado un grafo dirigido $G = (V, E)$ si para cada nodo $v \in V$ ocurre que $|\delta^+(v)| \geq 1$ entonces G contiene un circuito.

  : *Esta prueba es similar a la del grafo no dirigido, y la diferencia del grado del nodo radica en que en grafo no dirigidos, necesitas un grado de al menos dos para "entrar" y "salir" del nodo por arcos distintos. Port otro lado en el caso dirigido la condición de que $|\delta^+(n)| \geq 1$ es necesaria y suficiente.* ■

La idea de esta sección es la de dar algunas ideas en como argumentar pruebas y convencer al lector (o profesor) de que la prueba es correcta. Mostrando argumentos solidos, logicos y entendibles es la manera en la que la matemática está basada y una demostración mal argumentada puede conllevar a probar hechos que no son verdades.

En algoritmos III buscamos formalizar el conocimiento del estudiante y buscamos poder probar propiedades de ciertas estructuras así como también los algoritmos, tanto su correctitud como su complejidad. Un algoritmo sin una prueba formal de correctitud (que funciona) es simplemente un proceso que no garantiza la resolución del problema para el cual esta ideado.

3.2.2 Más tipos de grafos

Definición 3.2.1: Camino hamiltoniano

Un camino simple en un grafo $G = (V, E)$ es hamiltoniano si su longitud es $|V| - 1$

Definición 3.2.2: Ciclo hamiltoniano

Un ciclo simple en un grafo $G = (V, E)$ es hamiltoniano si su longitud es $|V|$

Definición 3.2.3: Camino euleriano

Un camino de un grafo G es euleriano si contiene todos los lados del grafo exactamente una vez, es decir que su longitud es $|E|$

Fíjese que en la definición de camino euleriano cualquier nodo puede estar repetido, lo importante es que el camino use todas las aristas.

Definición 3.2.4: Ciclo euleriano

Un ciclo en un grafo $G = (V, E)$ es euleriano si es un camino euleriano que empieza y termina en el mismo nodo

Aunque estas dos definiciones podrian parecer relacionadas, la verdad es que el problema de hayar un camino euleriano resulta ser mucho más sencillo que el de hayar un camino hamiltoniano. Esto tiene que ver con el hecho de que si un grafo tiene un camino euleriano basta con saber si cumple propiedades sencillas. Mientras tanto en el camino hamiltoniano resulta ser un problema de alta importancia en computación y en las matemáticas; encontrar un algoritmo que lo resuleva en tiempo polinomial (en el numero de nodos y arcos) o por el contrario demostrar que tal algoritmo no existe es un problema del milenio y resolverlo es premiado con un millón de dolares.
Pero por ahora, enfoquémonos un poco en los problemas mortales y empecemos por introducir algunas definiciones para poder demostrar que resolver el problema del camino euleriano es sencillo (que se puede resolver en tiempo polinomial).

Definición 3.2.5: Grafo euleriano

Un grafo conexo $G = (V, E)$ es euleriano si $\forall x \in V : |\delta(x)| \equiv_2 0$

Definición 3.2.6: Grafo semi-euleriano

Un grafo $G = (V, E)$ conexo es semi-euleriano si $\forall x \in V : |\delta(x)| \equiv_2 0$ excepto para exactamente 2 nodos.

Y con esto. cha chan chaaaaan llegamos al siguiente teorema

Teorema 3.2.7: *Un grafo es euleriano si y solo si existe un ciclo euleriano en él*

Primero que nada si un teorema dice "si y solo si", esto significa doble implicación, es decir equivalencia y hay que demostrar ambas direcciones.

: *Empecemos demostrando la parte mas sencilla primero.*

- ($\Leftarrow$) *Digamos que un grafo tiene un ciclo* C *que es euleriano. Observemos que cada vez que un nodo aparece en el ciclo es porque exactamente dos arcos fueron utilizados (para entrar y salir del nodo), esto quiere decir que si no nodo aparece* x *veces es porque tiene* $2 \cdot x$ *arcos adyacentes a él, exceptuando el primer nodo que es igual al último, en cuyo caso si el nodo aparece* y *veces es porque tiene* $2 \cdot (y-1)$ *aristas adyacentes a el. En ambos casos la cantidad de aristas adyacentes a los nodos es un multiplo de dos y esto quiere decir por definición es un grafo euleriano.*

- ($\Rightarrow$) *Ahora suponemos que el grafo es un grafo euleriano por lo cual también es conexo, tenemos que demostrar entonces que un ciclo euleriano existe dado que cada nodo tiene grado par. Hagamos una prueba constructiva, empecemos en un nodos cualquiera* n_1 *y hacemos recorridos en el grafo hasta volver a* n_1 *y veamos que volvemos a* n_1 *por el mismo argumento de que cada vez que entramos y salimos de un nodo es porque "usamos" dos arcos adyacentes a el mismo. Si en ese primer recorrido empezamos en* n_1 *entonces el recorrido sería* $n_1, e_1, n_2, ..., e_{k-1}, n_k$ *con* $n_k = n_1$ *ya que al salir inicialmente de* n_1 *el unico nodo en el recorrido que queda con grado impar es* n_1 *esto quiere decir que en los nodos intermedios siempre vamos a tener una manera de "salir" del nodo ya que su grado es par y siempre vamos a poder "salir" de los nodos hasta poder llegar de nuevo a* n_1*. En este primer ciclo que creamos no necesariamente hemos recorrido todos los arcos o nodos, sin embargo una propiedad que se cumple es que el número de arcos adyacentes no usados hasta ahora en cada nodo es par. Ahora supongamos que el ciclo inicial encontrado es* $C = (n_1, e_1, ..., e_k, n_1)$*, escogemos un nodo cualquiera digamos* n_i *que tiene aún lados disponibles (no utilizados en* C*) y hacemos el recorrido similar al proceso anterior pero esta vez empezando en* n_i *con lo que descubrimos un ciclo* $C' = (n'_1 = n_i, e'_1, ..., e'_{k'-1}, n'_{k'} = n_i)$*. Lo siguiente a hacer es agregar este ciclo a* C *de tal manera que queda el siguiente ciclo* $C'' = (n_1, e_1, n_2, ..., n_i = n'_1, e'_1, ..., e'_{k-1}, n_{k'} = n_i, ..., e_{k-1}, n_k$*. Repitiendo el mismo argumento construimos entonces logramos obtener un ciclo euleriano.* ■

Para el caso de grafos semieulerianos, tenemos un teorema muy similar.

Teorema 3.2.8: *Un grafo es semi-euleriano si y solo si existe un camino euleriano en que empieza y termina en nodos distintos*

:

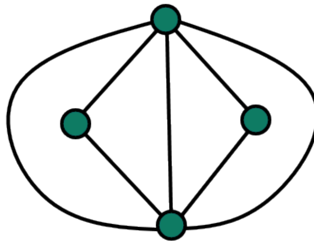
- ($\Leftarrow$) *Si existe un camino euleriano tal que empieza y termina en nodos distintos, esto quiere decir que el primer y último nodo tienen grado impar debido a el argumento de que cadavez que entramos y salimos de un nodo usamos 1 par de arcos. Sin embargo para el primer y último nodo, sus grados han de ser impar ya que en el primero "salimos" una vez mas de las que entramos y viceversa para el último nodo. El resto de nodos tienen un grado par por el mismo argumento del camino euleriano y "entradas" y "salidas".*

- ($\Rightarrow$) *Dado que el grafo es semi-euleriano, exactamente dos vertices del mismo tienen grado impar. Usamos un argumento similar a la prueba anterior con la exceptión de que la manera en que empezamos el recorrido es empezando de alguno de los dos nodos con grado impar. La manera en que ese recorrido terminará es llegando a el otro nodo de grado impar y podemos construir el resto del camino semi-euleriano repitiendo el mismo argumento de la prueba anterior* ■

Fijemonos que la manera en que hemos hecho ambas pruebas ha sido una manera constructiva. Lo bueno de algunas pruebas constructivas es que es posible derivar un algoritmo a partir de las mismas. En este caso la prueba lleva diretamente a un algoritmo que es incluso bastante eficiente y posible de implementar en tiempo lineal en $|V| + |E|$ para aun grafo $G(V, E)$.

Ejercicio 1

Encuentre un camino euleriano en el siguiente grafo (a).



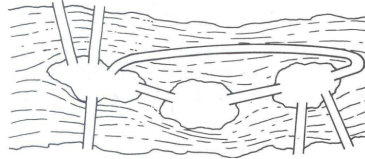
(a)

Ejercicio 2

Dibuje grafos mínimos eulerianos y semi-eulerianos

Ejercicio 3

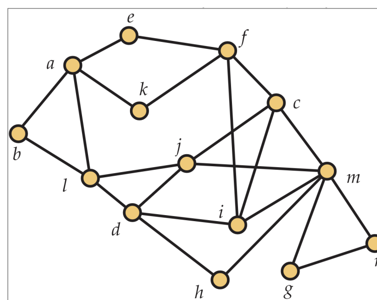
Represente la siguiente imagen en forma de grafo y diga si tiene un camino euleriano o no. ¿Y un ciclo euleriano?. Nota: Fijese que se puede entrar y salir por el lado mas exterior de la imagen en cualquier "entrada" del mismo.



(a)

Ejercicio 4

Diga si el siguiente grafo es euleriano y encuentre un circuito si lo es. Si no lo es explíque por qué.



(a)

3.3 Alcanzabilidad

Ya hemos hablado un poco por arriba sobre el camino entre dos nodos, pero vamos a formalizar ahora el concepto relacionado: **alcanzabilidad**.

Definición 3.3.1: Alcanzabilidad

Dado un grafo (dirigido o no) $G = (V, E)$, y dado $a, b \in V$, decimos que b es alcanzable desde a si existe un camino desde a hasta b.

Existe una diferencia muy sutil entre grafos dirigidos y no dirigidos. Si consideramos alzancabilidad como una relación, entonces la relación es simétrica para los grafos no dirigidos ya que si existe un camino uniendo dos nodos, entonces podemos usar el mismo camino para empezar en uno de esos dos nodos y alcanzar el otro. Además decimos que un codo es alcanzable desde si mismo (tomando un camino vaciío).

3.3.1 Matriz de alcanzabilidad

Dado un grafo $G = (V, E)$ quisieramos calcular una función en una matriz A de alcanzabilidad, cuya entrada $A_{i,j} = 1$ si y solo si para dos nodos $v_i, v_j \in V$ ocurre que v_j es alcanzable desde

v_i. Para construir la matriz de alcanzabilidad haremos un algoritmo constructivo basado en la siguiente proposición

Proposición 3.3.2

Existe un camino de largo k en un grafo no dirigido $G = (V, E)$ si y solo si $\{v, w\} \in E^k$ donde E^k es la composición de la relación E k veces y E^0 es la relación identidad. Además $\forall k \geq 1 : E^k \subseteq E \cup ... \cup E^n \subseteq E \cup ... \cup E^{n-1}$ donde $n = |V|$

: *Hacemos la prueba por inducción. Para $k = 0$ la proposición es obvia y si $k = 1$ entonces los caminos de largo 1 son los arcos.*

Ahora asumamos que la proposición es válida para $k \leq i - 1$ e intentamos demostrar para $k = i$. Dado un camino de largo i que existe en G, si asumimos que el camino es $(n_1, e_1, ..., e_i, n_{i+1})$ por inducción tenemos que $(n_1, e_1...., e_{i-1}, n_i) \in E^{i-1}$, por último ya que $e_i \in E$ entonces podemos construir el camino de longitud i que consideramos al principio. Por otro lado, para demostrar que $E^k \subseteq E \cup ... \cup E^n$ veamos que si $\{u, w\} \in E^k$ es porque existe un camino de largo k entre u y w si este camino es mas largo que n entonces existe un vertice repetido, por lo cual podemos removerlo y disminuir la longitud del camino. Repitiendo este argumento varias veces, vemos que la proposición se cumple ■

Proposición 3.3.3

Sea $G = (V, E)$ un grafo dirigido que no es multigrafo. Las siguiente propiedades son equivalentes.

- w es alcanzable desde v
- $(v, w) \in E^* = I \cup E \cup E^2 \cup ...$ donde I es la identidad sobre V
- El elemento $A_{i,j}$ de la matriz de alcanzabilidad es igual a 1

La prueba de la proposición la dejo de ejercicio.
Por último es posible probar de una manera simiilar a la primera proposición lo siguiente:

Proposición 3.3.4

$(v, w) \in E^*$ si y sólo si $(v, w) \in I \cup E \cup ... \cup E^{n-1}$, donde n=|V| el número de nodos el en grafo

Por otro lado, la matriz de adyacencia asociada a E^k la denotamos como $A(E^k)$ y nuetro objetivo final es poder calcular $A(E^*)$ es decir la matriz de alcanzabilidad sin importar la longitud de los caminos que ya sabemos que siempre será menor a n por las proposiciones anteriores.

3.3.2 Algoritmo de Roy-Warshall

A continuación denotaremos como M_k la matriz de alcansabilidad de hasta longitud k, es decir $M_k = A(E) + A(E^2) + ... + A(E^k)$. Basandonos en las proposiciones anteriores, el algoritmo primero calcula $M_1 = A(E)$, luego $M_2 = M_1 + M_1^2 = (I + M_1)M_1$, después $M_3 = M_1 + M_1^2 + M_1^3 = (I + M_1 + M_1^2)M = (I + M_2)M_1$, y generalizando llegríamos a $M_{n-1} = (I + M_{n-2})M$. Notemos que $A(E^*) = I + M_{n-1}$ y nuestro único problema para generar la matriz de adjacencia de un grafo

sería realizar operaciones entre matrices.

Multiplicar matrices con el método tradicional necesita tiempo $O(n^3)$ donde n son las dimensiones de una matriz cuadrada. Entonces para poder generar cada uno de los M_k necesitariamos tiempo $O(n^4)$ y la pregunta ahora es si podemos hacer esto un poco mejor. Cinsideremos la siguiente secuencia:

- $N_1 = I + A(E) = I + M_1 = I + M$
- $N_2 = N_1^2 = (I + M)^2$
- ...
- $N_{q+1} = N_q^2 = (I + M)^{2^q}$

Observando ahora que $A(E^*) = N_{q+1}$ para cierto q siendo el primer q tal que $n \leq 2^q$, entonces podemos acelerar el proceso a un tiempo de $O(n^3 \log_2(n))$.

Por otro lado, vamos a definir un algoritmo cuya correctitud probaremos a partir de ciertas observaciones y proposicones. Dado un grafo dirigido $G = (V, E)$ y $V = \{v_1, v_2, .., v_n\}$ definamos una relación binaria E_k sobre V, con $k \in [0, n]$ donde $(v, w) \in E_k$ si y sólo si existe un camino de v a w cuyo conjunto de vertices intermedios es un subconjunto de $\{v_1, ..., v_k\}$. Existen varias observacioines sencillas que podemos mencionar:

- Cuando $k = 0$, $\{v_1, .., v_k\}$ es el conjunto vacío
- $E_0 = I \cup E$
- $E_{k-1} = E_k$
- $E_n = E^*$

Luego podemos verificar la siguiente proposición

Proposición 3.3.5

Sea (v, w) tal que $v = v_k$ o $w = v_k$ con $1 \leq k \leq n$. Entonces, $(v, w) \in E_{k-1}$ si y sólo si $(v, w) \in E_k$

La prueba se deja como ejercicio. Y la de la siguiente proposición también

Proposición 3.3.6

Sea $(v, w) \notin E_{k-1}$. Entonces $(v, w) \in E_k$ si y sólo si $(v, v_k) \in E_{k-1} \wedge (v_k, w) \in E_{k-1}$

Juntando las proposiciones entonces podemos realizar un algoritmo calculando cada E_k a partir de E_{k-1} donde un nodo v alcanza a w en E_k si ya era alcanzable en E_{k-1} o si en nodo v_k alcanzable desde v en E_{k-1}, y w es alcanzable desde v_k en E_{k-1} o si $v = v_k \vee w = v_k$. Observando esto presentamos entonces el siguiente algoritmo

Algorithm 0: Roy-Warshall

```
M* ← I + M
for k ← 0 to n do
    for i ← 0 to n do
        if i ≠ k ∧ M*_{i,k} = 1 then
            for j ← 0 to n do
                M*_{i,j} ← M*_{i,j} + M*_{k,j}
```

Obviamente el tiempo de complejidad depende de como guardemos la matriz y que tan rápido accedemos a ella, pero ignorando ese hecho este algoritmo terminaría teniendo una complejidad de $O(n^3)$, justifique.

3.3.3 Clausura transitiva de un grafo

3.4 Componentes conexas

Vamos a formalizar ahora el concepto de componentes fuertemente conexas y componentes conexas, pero primero observemos que la relación de alcanzabilidad mutua es una relación de equivalencia, es decir, dado dos nodos v, w, si v es alcanzable desde w y viceversa, entonces podemos decir que ellos son equivalentemente alcanzables. Esto tiene sentido ya que una vez alcanzado v entonces podemos decir que w es igual de alcanzable que v ya que existe un camino entre ellos dos.

Definición 3.4.1: $\equiv_{alc}$

Dado un grafo cualquiera, decimos que dos nodos $v, w \in V$ son equivalentemente alcanzables, o $v \equiv_{alc} w$, si es posible alcanzar w desde v y viceversa.

Definición 3.4.2: Componente fuertemente conexa

Dado un grafo dirigido $G = (V, E)$, y sus clases de equivalencia de alcanzabilidad a las que denotamos como $V_1, V_2, ..., V_k$, las componentes fuertemente conexas son aquellos grafos inducidos por $V_1, V_2, ..., V_k$, es decir $G[V_1], ..., G[V_k]$

Definición 3.4.3: Componente conexa

Es analogo a la definición anterior pero para grafos no dirigidos. En este caso, si dos nodos estan conectados por algun camino, entonces son equivalentes.

Definición 3.4.4: Grafo conexo

Un grafo es conexo si su numero de componentes conexas es 1. En el caso dirigido, el grafo es conexo si su grafo no orientado asociado es conexo.

Proposición 3.4.5

Dado un grafo dirigido, demuestre que un camino entre dos vertices de una componente fuertemente conexa existe tanto en el grafo original como en el grafo inducido de la componente.

Por otro lado, veamos un concepto relacionado: **grafo reducido**.

Definición 3.4.6: Grafo reducido

Digamos que un grafo G tiene las componentes fuertemente conexas $G[V_1], ..., G[V_k]$. El grafo reducido R_G de G es un grafo cuyos nodos son las componentes fuertemente conexas del grafo original, a estos nodos los podemos denominar $V_1, ..., V_k$ y existe un arco (V_i, V_j) en R_G si y sólo si existe un arco (v_i, v_j) en G con $v_i \in V_i \wedge v_j \in V_j$

Dada la definición anterior, demuestra las siguientes proposiciones

Proposición 3.4.7

Un grafo reducido de un grafo dirigido no posee circuitos

Definamos para cada nodo v en un grafo $G(V, E)$ los **descendientes** de v como aquellos nodos alcanzables desde v y denominemos a este conjunto $D(v)$. De igual manera, el conjunto de nodos que alcanzan a v lo llamamos el conjunto de **ascendientes** y lo denotamos por $A(v)$. Demuestre la siguiente proposición

Proposición 3.4.8

Dado un grafo dirigido $G = (V, E)$ y $v \in V$, los vertices de la componente fuertemente conexa que contiene a v que denotaremos como $C(v) = D(v) \cap A(v)$. Para un grafo no dirigido ocurre que $C(v) = A(v) = C(v)$

3.4.1 Calculo de componentes fuertemente conexas

Existen varios algoritmos para calcular componente fuertemente conexas, fijémonos que anteriormente las hemos definido en función de la intersección de dos conjuntos donde dado un nodo cualquiera simplemente debemos intersectar el conjunto de vertices que ese nodo alcanza y el conjunto de nodos que a ese vertice lo alcanzan. Anteriormente vimos como es posible calcular la matriz de alcanzabilidad en donde podemos observar que nodos son alcanzables desde un nodo en particular, con esto podemos calcular $D(v)$ para un nodo v. Sin embargo, aún tenemos el problema que no sabemos calcular $A(v)$, una manera de hacerlo es revirtiendo todos los arcos dirigidos del grafo y calculando la matriz de alcanzabilidad en el grafo inverso G^{-1}. Una vez esto hecho, entonces solo debemos ver que nodos son alcanzables para v tanto en G como en G^{-1} y ese conjunto es entonces una componente fuertemente conexa.
Aunque es posible calcular dos veces una matriz de alcanzabilidad, existe una proposición que nos ayudara a computar la segunda vez la matriz de alcanzabilidad.

Proposición 3.4.9

Sea A^* la matriz de alcanzabilidad de un grafo dirigido (o no) G, y sea G^{-1} como definimos anteriormente. Entonces $A^*_{G^{-1}} = (A^t)^* = (A^*)^t$

Demuestre la proposición anterior.
Gracias a la proposición entonces vemos que solo es necesario calcular la matriz de alcance una vez y con esto llegamos a nuestro algoritmo para calcular componentes fuertemente conexas.

Algorithm 1: Cálculo de componentes fuertementes conexas 1

```
Input: G = (V, E)
Calcular matriz de alcance A* del G
S ← V
Componentes ← ∅
while S ≠ ∅ do
    v ← S
    C ← {v}
    for w alcanzable desde v do
        if v alcanzable desde w then
            C ← C ∪ {w}
    Componentes ← Componentes ∪ C
    S ← S \ C
```

Para analizar el tiempo de complejidad del algoritmo veamos que el primer paso tiene un tiempo de complejidad de $O(|V|^3)$ por el algoritmo visto en la sección anterior para calcular la matriz de alcanzabilidad. El resto de complejidad viene dada por el numero posible de componentes conexas en un grafo que esta acotado superiormente por $|V|$ y la cantidad de operaciones del bucle interno que es a lo sumo $|V|$ esto quiere decir que la complejidad de las operaciones (aunque depende del tipo de estructura que usamos para guardar los conjuntos) al calcular las componentes es $O(|V|^2)$, por lo tanto nuestro algoritmo tiene una complejidad de $O(|V|^3)$. Mas adelante veremos que existe un algoritmo lineal en el numero de nodos y arcos para calcular las componentes fuertementes conexas, sin embargo necesitamos aprender primero un poco sobre recorridos en grafos.

3.5 Conjuntos de articulación

Ahora entramos en un nuevo tema asociado a componentes conexas: **puntos de articulación** o **Conjuntos de articulación**. Ya hemos visto las definiciones de grafo conexo / disconexo tanto en el caso orientado como en el no orientado, donde dijimos que si tomamos el grafo de manera no orientada y no hay mas de una componente conexa, entonces el grafo es conexo. Con esto llegamos a la siguiente definición

Definición 3.5.1: Conjunto de articulación

Si $G = (V, E)$ es un grafo fuertemente conexo, un conjunto $V' \subset V$ es un **conjunto de articulación** si $G[V - V']$ es disconexo.

Definición 3.5.2: Conjunto de articulación para grafos no dirigidos

Si $G = (V, E)$ es un grafo fuertemente conexo, un conjunto $V' \subset V$ es un **conjunto de articulación** si $G[V - V']$ contiene al menos una componente conexa más que en el grafo original.

Definición 3.5.3: Punto de articulación

Dadas las dos definiciones anteriores, un punto de articulación es lo que ocurre cuando $|V'| = 1$, es decir si $|V'| = \{v\}$ para algun $v \in V$ entonces v es un punto de articulación

Definición 3.5.4: Conectividad de un grafo

Es el tamaño mínimo de un conjunto de verticies de un grafo que al ser eliminados desconectan al mismo o queda un solo nodo, se de nota $\kappa(G)$ para un grafo dado G

Definición 3.5.5: Grafo h-conexo

Un grafo es $h-conexo$ si $h \leq \kappa(G)$

Definición 3.5.6: Puente

Un puente es el concepto análogo a puntos de articulación pero para lados, es decir, una arista es un puente si al ser removida el número de componentes conexas (o fuertemente conexas) del grafo aumenta en al menos una unidad.

Demuestre si la siguiente proposición es verdadera o falsa

Proposición 3.5.7

La existencia de puentes en un grafo con un mínimo de 3 verticies implica la existencia de puntos de articulación

Intente derivar un algoritmo para calcuar puntos de articulación (sin importar su complejidad). En los próximos capítulos veremos algoritmos bastante eficientes para calcular puntos de articulación y puentes.

Demuestra la siguiente proposición

Proposición 3.5.8

Un grafo es conexo si la matriz de adyacencia asociada es una matriz de solamente 1s.

Definición 3.5.9: Cociclo asociado

Dado $S \subseteq V$ para un grafo $G = (V, E)$ el cociclo asociado de S es $\Omega(S) = \{e = \{u, v\} : e \in E \wedge |e \cap S| = 1\}$

Esta definición simplemente indica el conjunto de arcos que separan a un conjunto de vertices S de su complemento $\bar{S} = V - S$. La definición es análoga para grafos dirigidos, definiendo $\Omega^{+}(S)$ y $\Omega^{-}(S)$.

Demuestre la siguiente proposiciones

Proposición 3.5.10

Dado un grafo $G = (V, E)$. La condición necesaria y suficiente para que G sea conexo (fuertemente conexo) es que, para todo conjunto propio S de V se tenga que $\Omega(S) \neq \emptyset$ $(\Omega^{+}(S) \neq \emptyset)$

Proposición 3.5.11

Todo grafo conexo de al menos dos nodos posee al menos dos vertices que no son puntos de articulación

Proposición 3.5.12

En un grafo no dirigido un nodo de un grafo es un punto de articulación si existen dos nodos distintos tales que todo camino entre ellos pasa por el punto de articulación.

Proposición 3.5.13

Un lado de un grafo es un puente si no pertenece a ningún ciclo del grafo

Proposición 3.5.14

Si un grafo es un árbol entonces existe un y sólo un camino simple entre cada par de vertices.

SECCIÓN 4

Recorrido de grafos

Hasta ahora hemos visto un algunos algoritmos cuya implementación no está tan clara debido al uso de estructuras bastante abstractas como el uso de conjuntos que en la realidad no siempre son fáciles de implementar con operaciones que tomen un tiempo razonable. En esta sección nos adentramos un poco más en el mundo de los grafos para aprender como recorrerlos de diferentes maneras y las ventajas que cada una de ellas presenta. En esta sección nos empezaremos a interesar por el problema de encontrar un camino entre dos vertices de un grafo que luego deriva y ayuda de base para la construcción de nuevos algoritmos para otro tipo de problemas. Además empezaremos a entender en mas detalles algoritmos para calcular puntos de articulación, puentes, y de allí seguiremos con temas mas avanzados en las siguientes secciones.

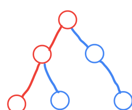
4.1 El recorrrido de grafos

En el recorrido de grafos nos interesamos no solo en recorrer el grafos de alguno forma, sino también de dejar cierto tipo de información en cada nodo que nos ayude a determinar cierta información, por ejemplo es posible dejar como información en que momento un nodo ha sido visitado, que tan lejos esta desde otro nodo, si el nodo ha sido visitado o no, etc. En los algoritmos que veremos proximamente veremos que dependiendo del tipo de información o etiqueta que vayamos dejando podremos probar propiedades del grafo que derivan en mejoramiento del tiempo de complejidad o espacio.

Además no haremos diferenciación de los algoritmos para grafos dirigidos o no, ya que la mayoría aplican para ambos casos y en el caso de discernir entonces será anunciado explicitamente.

4.2 Búsqueda en profundidad o Depth First Search (DFS)

Este algoritmo es el mas intuitivo de generar, como su nombre indica estamos interesados en hacer una búsqueda lo mas profundo posible en un grafo, es decir si un nodo es "visible" en cierto nivel de un grafo, decidimos recorrer aquel nodo cuya profundidad es mayor. Veamos un ejemplo



(a) DFS, búsuqeda en profundidad marcada en rojo

En la figura anterior, digamos que el nodo que está mas arriba es el nodo donde empezamos algun tipo de búsqueda. Luego de haber visitado el nodo raíz, en lugar de expandir ambos hijos lo que se observa en la figura es que se expande el hijo "izquierdo", y luego nuevamente en ese hijo izquierdo en lugar de expandir ambos hijos seguimos expandiendo la búsqueda por su hijo izquierdo también. Este es el significado de la búsqueda en profundidad, buscar lo más profundo posible en lugar de buscar ampliamente. La manera más fácil de implementar este algoritmo es mediante una función recursiva que vaya marcando los nodos visitados.

Algorithm 1: DFS

Input:

$raiz \in V$

$G = (V, E)$

$visited[raiz] \leftarrow true$

Hacer algun procesamiento de $raiz$

for $\{raiz, ady\} \in \delta(raiz)$ **do**

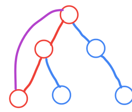
  if $!visited[ady]$ **then**

    $DFS(ady)$

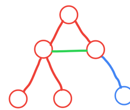
Lo que aún es abstracto en el algoritmo anterior es la representación o la manera en que recorremos los nodos adyacentes. Dado que esta recursión visita solo los nodos que aún no han sido visitados, existen a lo sumo $O(|V|)$ llamadas a DFS, el resto de tiempo de complejidad viene dado por la manera de recorrer los lados adyacentes (como en el primer capítulo). Si la representación es hecha mediante lista de adyacencias entonces el tiempo total de DFS sería $O(|V| + |E|)$ mientras que si lo hacemos mediante matriz de adyacencia entonces el tiempo de complejidad sería $O(|V|^2)$. Por otro lado, para grafos digiridos la implementación es similar, y lo único que cambiaría es que usamos $\delta^+(raiz)$ en lugar de $\delta(raiz)$.

Por otro lado, podemos definir el **padre** de un nodo v en el grafo, como el primer nodo que "descrubrió" a v, es decir el nodo adyacente a v que hizo la llamada $DFS(v)$. Esta llamada implicaría entonces que v no había sido visitado anteriormente desde otro nodo del grafo. Esto quiere decir que el nodo padre de un nodo cualquiera v es el nodo raíz que llamo a DFS para v

en el "if" interno del algoritmo presentado anteriormente.
Además podemos claisifcar los arcos que vamos descubriendo durante nuetro DFS que usaremos próximamente para desarrollar otro tipo de algoritmos.



(a) Lado de retorno (marcado en morado), es un lado que va desde un nodo que recien está siendo visitado hacia un nodo ancestro ya visitado



(b) Lado cruzado (marcado en verde), es un lado que va desde un nodo que recien está siendo visitado hacia un nodo que pertenece a una rama distinta del árbol de búsqueda

Además existen los lados forward que simplemente llamaremos lados, que son los lados marcados en rojo. Estos son lados que descubren nuevos nodos del grafo.

Definición 4.2.1: Árbol DFS

El árbol DFS es simplemente el árbol formado por todos los lados **forward** del recorrido DFS en un grafo conexo.

Definición 4.2.2: Arborescencia DFS

Cuando el grafo no es conexo, podemos llamar DFS en un nodo en cada una de las componentes conexas del grafo para recorrerlo enteramente. En este caso el recorrido DFS no formaría un árbol sino un arborescencia, la **arborescencia DFS**

Dadas todas estas definiciones, que aplican para grafos dirigidos o no, entonces ahora procedemos a preguntarnos ¿Cómo es posible detectar in ciclo?, la respuesta es bastante sencilla: mediante el uso de Lados de retornos (back edges). Demuestre la siguiente proposición

Proposición 4.2.3

En un grafo no dirigido no existen cross edges (lados cruzados)

Ahora demostremos la siguiente proposición

Proposición 4.2.4

Un grafo $G = (V, E)$ (dirigido o no) tiene un ciclo $\iff$ su arbol de DFS tiene un backward edge

:

- *(⇒) Partimos de que el grafo G tiene un ciclo* $C = (v_1, e_1, v_2, e_2, ..., e_k, v_1)$*, esto quiere decir que cuando hagamos un DFS sobre él existe un nodo de ese ciclo que será el primero en ser alcanzado, supongamos que es* v_1*. Dado que es el primer nodo del ciclo que sería visitado por el DFS entonces los nodos de C van a ser recorridos como sucesores de* v_1 *ya sea en el mismo orden de* C *o en algun otro orden, de cualquier manera* v_{k-1} *va a ser alcanzado como succesor de* v_1 *en algun punto y va a llamar gracias al arco* e_k *a* $DFS(v_1)$*, que ya ha sido visitado y por lo tanto esto es un lado de retorno en el árbol (o arborescencia) del* DFS*.*
- *(⇐) Por otro lado, si existe un backward edge que asumimos sale del nodo u al nodo* v*, entonces si tomamos todos los nodos ancestros de u hasta llegar a* v *(que es ancestro también, de lo contrario el edge no sería backward) entonces todos estos nodos forman un ciclo* ■

En conclusión detectar un ciclo en un grafo es tan difícil como detectar un backward edge, es decir, son problemas equivalentes. Ahora ya sabemos como calcular ciclos en grafos dirigidos y no dirigidos.

4.2.1 Implementación iterativa de un DFS

Ya dijimos anteriormente como sería la implementación recursvia de un DFS, con esto nos podemos hacer la pregunta si es posible de alguna forma de hacerlo de manera iterativa. Plot twist: sí, todo proceso recursivo puede ser implementado de manera iterativa y la razón de esto es que una solución recursiva al final usa en realidad estructuras de datos al ser imlpementadas en la computadora y la estructura que ella hace es llamada el **call stack** o pila de llamadas, que guardan información en cada momento sobre variables locales de cada nivel de recursión por ejemplo. De igual manera podríamos hacer una simulación de esto haciendo uso de una estructura que ya conocemos de Algoritmos II: **La pila**.

Algorithm 1: DFS iterativo

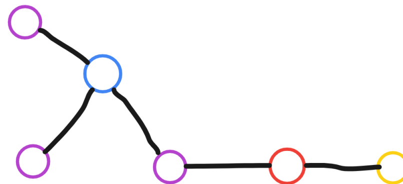
Input:
$raiz \in V$
$G = (V, E)$

$pila \leftarrow (raiz)$
while *pila no vacia* **do**
  $v \leftarrow pila.pop()$
  $visited[v] \leftarrow true$
  Hacer algun procesamiento de v
  for $\{raiz, ady\} \in \delta(v)$ **do**
    if $!visited[ady]$ **then**
      pila.append(ady)

El tiempo de complejidad sería el mismo que el del algoritmo recursvio con la salvedad de que no existe el call stack que mencionabamos anteriormente.

4.3 Búsqueda en amplitud o Breadth First Search (BFS)

Hasta ahora hemos visto un solo algoritmo para recorrido de grafos (DFS) que intenta buscar siempre el nodo mas profundo, sin embargo existe otra forma de recorrer los grafos y es por amplitud, esto es, empezamos recorriendo algun nodo raíz, y luego recorremos consecuentemente todos los nodos que son directamente hijos de la raíz. Luego, todos aquellos nodos a un lado de distancia de los nodos anteriores y así sucesivamente. La diferencia radica entonces en que en BFS recorreriamos el grafo por niveles de separación desde el nodo raíz. Veamos esto un poco mejor en la siguiente figura



(a) BFS: búsuqeda en amplitud. El nodo azul es el nodo raíz. Los nodos en morado representan 1 grado de separación desde la raíz. El nodo rojo esta a 2 grados de separación y el amarillo a 3.

En el caso de BFS es posible dividir los nodos por niveles (o grados de separación) y esto nos

provee propiedades importantes que veremos luego de presentar formalmente el algoritmo pero primero notemos que existe una estructura de dato que es perfecta para la implementación de un BFS: **la cola o queue**.

Algorithm 1: BFS

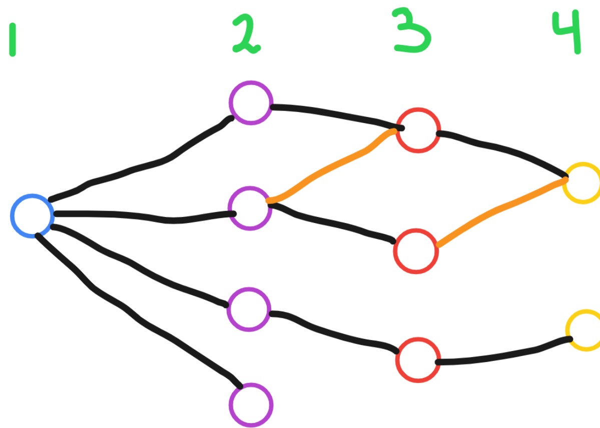
Input:
 $raiz \in V$
 $G = (V, E)$
$cola \leftarrow (raiz)$
$level[raiz] \leftarrow 0$
while *cola no vacia* **do**
 $v \leftarrow cola.pop()$
 $visited[v] \leftarrow true$
 Hacer algun procesamiento de v
 for $\{raiz, ady\} \in \delta(v)$ **do**
  if $!visited[ady]$ **then**
   cola.append(ady)
   $level[ady] \leftarrow level[v] + 1$

Primero que nada, es claro que la cantidad de nodos que pueden estar dentro de la cola al mismo tiempo es $O(|V|)$, y recorrer el grafo dado que tenemos la lista de vertices visitados tomaría tiempo $O(|V| + |E|)$. Existe una pequeña ventaja de DFS contra BFS y es que en un grafo que es que en grafos densos lo mas probable es que BFS termine metiendo muchisimos nodos a la cola, mientra que la cantidad de nodos en una pila de un DFS está acotada por el camino mas largo en el grafo. Para ver esto con mas detalle, supongamos que queremos analizar el problema del cubo de Rubic y lo logramos representar en un grafo donde dos estados estan unidos por un arco (no dirigido) si se puede llegar de un estado al otro en un solo paso. El factor de ramificación de este grafo es de 4, es decir cada nodo tiene 4 posible sucesores, esto quiere decir que en el nivel d de profundidad del árbol BFS pueden haber tanto como 4^d nodos que pueden estar al mismo tiempo en la cola del BFS. Por otro lado, en un DFS no es necesario guardar todos los nodos, si el grafo tiene una profundiad de d entonces a lo sumo d nodos estarán al mismo tiempo siendo considerados en la recursión del DFS.

Con esto nos podemos preguntar entonces para qué necesitamos BFS, veamos la arborescencia que se formaría al ejecutar $BFS(v)$ para un nodo cualquiera v. La imagen 10 es casi una arborescencia (ya que contiene ciclos) donde conservamos los arcos en naranja para poder ilustrar que son arcos que pudieron haber "descubierto" el nodo a su derecha. Los números en verde indican el "nivel" de cada nodo y una primera observación que podemos hacer es que entre nodos que difieren de mas de un nivel es imposible que haya un arco y es lo que observaremos a continuación.

Proposición 4.3.1

Dada la arborescencia BFS de un grafo no dirigido $G = (V, E)$, si asignamos a cada $v \in V$ una etiqueta $level(v)$ que indica grado de separación del nodo raíz $r \in V$, entonces para dos nodos $x, y \in V : |level(x) - level(y)| > 1 \Rightarrow \{x, y\} \notin E$



(a) BFS: búsuqeda en amplitud. El nodo azul es el nodo raíz. Los nodos en morados representan 1 grado de separación desde la raíz. El nodo rojo esta a 2 grados de separación y el amarillo a 3.

Figure 10: Ejemplo BFS

: *Asumamos que existen dos nodos en la arborescencia $x, y \in V$ cuyos niveles difieren por mas de una unidad, sin perdidas de generalidad asumamos que $level(x) < level(y)$ y además $level(y) > level(x) + 1$. Por reducción al absurdo, asumamos que existe un arco $\{x, y\} \in E$, esto quiere decir que en el momento en que hagamos un BFS desde el nodo raíz r, por hipoótesis x sería descubierto primero a $level(x)$ grados de separación de r, luego por el algoritmo de BFS cuando el nodo x es procesado todos los nodos adyacentes no visitados anteriormente serán agregados a la cola con un nivel más que el del nodo x, esto quiere decir que en particular y sería agregado (en caso de no haber sido visitado anteriormente) con $level(y) \leq level(x) + 1$* ■

Con esto podemos ahora pensar en una conclusión aún mas fuerte y podemos preguntarnos si para un nodo $x \in V$ ocurre que $level(x)$ es exactamente el camino mínimo (cantidad mínima de lados recorridos) desde la raíz del BFS hasta llegar a x y lo cierto es que sí y lo probaremos a continuación.

Proposición 4.3.2

Dada un recorridog BFS de un grafo dirigido o no $G = (V, E)$, si asignamos a cada $v \in V$ una etiqueta $level(v)$ que indica grado de separación del nodo raíz $r \in V$ como lo hace el mismo algoritmo de BFS, entonces $level(x)$ es el número mínimo de arcos a recorrer hasta llegar a x desde la raíz

: *Procederemos por inducción en el número mínimo de arcos a recorrer hasta llegar a cualquier nodo del grafo:*

- *Para el nodo raíz r que es el único nodo cuyo grado de separación desde la misma raíz es 0 (está en las primera lineas del algoritmo BFS) tenemos que $level(r) = 0$ que es exactamente la cantidad mínima de lados para un camino desde r a r. De igual forma ocurre para los nodos directamente adyacentes a r cuyo $level$ sería uno (por la primera iteración del algoritmo)*

- *Asumimos ahora que la proposición es cierta para todos los nodos $x \in V$ con $level(x) \leq p$ y demostramos la proposición para nodos $y \in V$ con $level(y) = p+1$ luego de haber ejecutado $BFS(r)$. Por la proposición anterior, sabemos que no existe un arco entre un nodo $z \in V$ con $level(z) < level(x)$ e y, de lo contrario $level(y) \neq p+1$. De la misma manera no existe un camino de tamaño menor a $p+1$ que alcance a y, de lo contrario asumamos que ese camino es $C = (v_0, e_0, ..., e_{k-1}, v_{k-1}, e_{k-1}, y)$ con $k < p+1$, esto quiere decir que existe un arco $\{v_{k-1}, y\} \in E$ donde $level(v_{k-1}) \leq k-1 < p$ (por hipótesis). Luego, como $level(v_{k-1}) < p$ quiere decir que BFS pudo haber alcanzado a y mediante e_{k-1} y asignar $level(y) = k < p+1$ contradiciendo lo que se asumia inicialmente sobre y* ■

Lo mas importante a rescatar es que con una sola llamada a BFS desde alguna raíz encontramos la longitud del camino desde la raíz a cada nodo del grafo y el mismo análisis aplica para grafos dirigidos. Por otro lado, es fácil recuperar un camino de tamaño mínimo a cada nodo del grafo asignando un **padre** a cada nodo que sería el nodo que lo incluye en la cola como parte del algoritmo del BFS, hacer este algoritmo con asignación de padres y una función para recuperar el camino se deja como tarea.

4.4 Ordenamiento topológico

Un problema común a la hora de resolver grafos es el del **ordenamiento topológico** en donde existen un conjunto de tareas y dependencias entre ellas indicando cuál debe ser estrictamente ejecutada primero y el problema es buscar un orden de ejecución de las tareas que cumplan con las dependencias mencionadas. En otras palabras dada $|T| = n$ tareas y $|E| = m$ dependencias entre ellas de la forma $(x, y) \in E \subseteq T \times T$ debemos encontrar un orden de ejecución tal que $\forall (x, y) \in E$, x se ejecuta antes que y.
Este problema puede ser representado entonces como un grafo dirigido de n nodos y m lados, y fijémonos que en el caso de existir un ciclo en este grafo entonces el problema no tendría solución. Por otro lado, ya tenemos dos herramientas que serían el BFS y el DFS para intentar resolver este problema, sin embargo veremos a continuación el algoritmo de Kahn que intenta hacer un algoritmo parecido a un DFS con ciertas particularidades donde la primera observación a hacer es que todas aquellas tareas que no tienen ninguna tarea precedente pueden ser ejecutadas inmediatamente. A partir de esa observacion podemos quitar entonces esas tareas sin dependencias y incluir en una cola las tareas sucesoras y proceder a ejecutar luego aquellas tareas sucesoras que tengan dependencias que ya hayan sido ejecutadas y finalmente se itera sobre esta misma idea. Veamos entonces el algoritmo

Algorithm 2: Kahn (1962)

Input:

$G = (V, E)$

$S \leftarrow$ conjunto de nodos sin precedencias (sin arcos de entrada)

$L \leftarrow ()$ // Lista vacia que contiene el orden topologico

while $S \neq \emptyset$ **do**

$n \leftarrow S.pop()$

$L.append(n)$

for $(n, ady) \in \delta^+(n)$ **do**

$E = E - \{(n, ady)\}$

if $|\delta^-(ady)| = 0$ **then**

$S \leftarrow S \cup \{ady\}$

if $|E| \neq 0$ **then**

return *Error, el grafo tiene un ciclo*

return L

La pregunta ahora es saber si este algoritmo es correcto y saber que complejidad tiene. Asumiendo que podemos "remover" un arco en tiempo constante, notamos que su tiempo es lineal en el número de arcos y nodos, ya que cada nodo sería procesado una vez por el while loop y cada arco sería removido y tomado en cuenta una sola vez. Por otro Lado, saber si el algoritmo es correcto, es decir, calcula un orden topológico es un poco mas coplicado.

Proposición 4.4.1

El algoritmo 6 computa un orden topológico de un grafo, o retorna error en el caso de ser imposible.

: *El arugmento para saber si el algoritmo es correcto o no viene dado por la misma idea del parrafo inicial de esta sección. Inicialmente S puede estar vacío o no. En el caso de estar vació quiere decir que todo nodo del grafo tiene algun precedente (arco incidente sobre él) esto solo quiere decir que no existe una primera tarea que pueda ser ejecutada inicialmente, entonces el algoritmo retornaría error lo cual es correcto. Si S no es vacío entonces el algoritmo en el ciclo pasa a "ejecutar" todas aquellas tareas que no tienen precedencias o cuyas dependencias ya hayan sido ejecutadas. Esto quiere decir que cuando el tamaño de L aumenta es porque es correcto ejecutar una tarea y no estamos cometiendo ninguna infracción de ejecutar alguna tarea anterior a sus dependientes. Esto se debe a que al quitar arcos de E estamos habilitando nuevas tareas a ser integradas en S una vez que todas sus dependencias hayan sido ejecutadas lo que hace que no se violen las dependencias de las tareas en E.*

Finalmente cuando el while loop termina es porque S está vacío, y existen dos opciones, que aun existan arcos en E o no. Si no existe ningun arco en E es simplemente porque ya todas las tareas fueron ejecutadas y entonces L contiene un orden topológico válido gracias a la idea expresada en el párrafo anterior. Por otro lado si $|E| > 0$ *es porque existe al menos un arco* $(x, y) \in E$ *que no ha sido removido, la única razón de esto es que x no haya sido ejecutada ya que aún tiene dependencias no ejecutadas y sus dependencias al mismo tiempo tienen dependencias, esto sólo puede ocurrir por la existencia de un ciclo en alguna parte del grafo con lo cual el algoritmo de Kahn retorna el error esperado.* ■

4.5 Algoritmo de Tarjan para componentes fuertemente conexas

Este algoritmo esta basado en la observacion que vimos anteriormente con respecto a encontrar backward edges para detectar componentes fuermente conexas, además usa una técnica de recorrido de grafos que ya hemos visto: DFS.

Por otro lado, para la búsqueda de backward edges, el algoritmo utiliza etiquetamiento, es decir, cada nodo al ser visitado recibe dos tipos de etiquetas que serán utilizadas para saber el tipo de arco que nos encontramos al salir de cada nodo. Cada nodo $x \in V$ será etiquetado con dos números:

- $low[x]$: un número que indica el menor discovery time de un nodo que puede ser alcanzado en el subárbol de DFS de x.
- $disc[x]$: un número que indica cuándo un número fue descubierto. Para el nodo raíz del DFS su discovery time es 0, para su primer sucesor sería 1, para el primer sucesor de ese sucesor sería 2,...

Para cada nodo recientemente descubierto el algoritmo asigna $low[x] \leftarrow disc[x]$. Luego, si existe un arco $(x, y) \in E$ existen dos casos

- y no haya sido visitado antes en el DFS, en este caso $low[x]$ puede ser actualizado a $\min(low[x], low[y])$
- y ha sido visitado antes, entonces actualizamos $low[x]$ a $\min(low[x], disc[y])$

Finalmente, con esta información una manera de detectar que existe una componente fuertemente conexa es mediante el uso de $low[x]$. Todos los nodos cuyo low es distinto a su $disc$ pertenecerían a alguna componente fuertemente conexa y es un hecho que probaremos luego de presentar el algoritmo de Tarjan donde presentaremos una subrutina que tiene que ser ejecutada para cada nodo que aun no haya sido visitado.

Algorithm 3: componentes fuertemente conexas de Tarjan (1972)

```
Rutina: Tarjan
Input:
    raiz ∈ V
    G = (V, E)
    Q una pila para ir guardando los nodos
    visited un arreglo de nodos visitados
disc[raiz] ← time++ // asignamos un discovery time a la raiz
low[raiz] ← disc[raiz]
visited[raiz] ← true
Q.push(raiz)
for (raiz, ady) ∈ δ+(raiz) do
    if !visited[ady] then
        Tarjan(ady)
        low[raiz] ← min(low[raiz], low[ady])
    else
        // Verificamos que no sea un cross-edge
        if ady ∈ Q then
            low[raiz] ← min(low[raiz], disc[ady])
//Creamos nueva componente fuertemente conexa
if low[raiz] = disc[raiz] then
    C ← ∅
    while Q.top() ≠ raiz do
        C ← {Q.top()}
        Q.pop()
    C ← {raiz}
    Q.pop()
```

Para probar la correctitud del algoritmo de Tarjan presentamos entonces la siguiente proposición que de ser correcta, prueba directamente la correctitud del calculo de las componentes fuertemente conexas.

Proposición 4.5.1

Dado un grafo dirigido $G = (V, E)$ en el algoritmo de Tarjan todos los nodos con un mismo valor low pertenecen a la misma componente conexa

:

Supongamos que existe una componente fuertemente conexa $C = \langle n_1, ..., n_k \rangle$ con $n_1, ..., n_k \in V$ que no es subcomponente conexa de otra, es decir es lo más grande posible. Ahora supongamos que en la llamada a $Tarjan$ los nodos son visitados en este orden de C con lo que $disc[n_1] < disc[n_2] < ... < disc[n_k]$, pero dado que esto es un circuito existe un arco desde $n_k \in C$ a un otro $n_j \in C$ de menor discovery y luego desde n_j a un $n_{j'} \in C$ y así sucesivamente hasta poder llegar de regreso a n_1. Esto quiere decir, que el único nodo para el cual $disc[u] = low[u]$ es para $u = n_1$. Es decir que ninguno de los nodos en C han sido extraídos de Q para el momento en que la llamada recursiva vuelva a $Tarjan(n_1)$, dado que C no puede ser expandido a una componente mas grande tenemos que $disc[n_1] = low[n_1]$, por lo que el algoritmo procedera a extraer de Q todos los nodos en C, incluyendo n_1 y la componente fuertemente conexa creada sería justamente C. ■

Notemos que el teorema anterior no implica que todos los nodos de una misma componente fuertemente conexa tienen el mismo low ya que esto no ocurre para circuitos dentro de circuitos. Finalmente nos falta demostrar que el algoritmo es correcto, es decir que computa todas las components fuertemente conexas.

Proposición 4.5.2

Dado un grafo dirigido $G = (V, E)$ en el algoritmo de Tarjan computa todos los circuitos máximos (sin contar subcircuitos) de manera correcta que son alcanzables desde la raiz dada

:

La prueba viene dada por dos observaciones, la primera es que todo nodo alcanzable desde la raíz será pusheado en la pila Q ya que Tarjan es un DFS con algunas modificaciones que no bloquean el recorrido.
La segunda observación es que supongamos que existe un circuito máximo $C = (n_1, ..., n_k)$ de k nodos y sea n_i el primer nodo alcanzado en el algoritmo de Tarjan desde la raíz. Llamemos a n_i el representante de C, sea $low[n_i]$ su low. Para todo nodo $n_j \in C$ se cumple que al finalizar el recorrido $low[n_j] \geq disc[n_i]$, de lo contrario existe un arco a un predecesor de n_i en el arbol de DFS del algoritmo y C dejaría de ser máximo.

Por otro lado, el único nodo en C al final del algorimo cuyo low es igual a su discovery es n_i ya que de lo contrario existe un nodo que no alcanza a n_i lo que es falso ya que C es un circuito. Incluso en el caso de haber subcircuitos ocurre este hecho al aplicar este argumento para ciclo simples y luego agregando subcircuitos. Por lo tanto, todos los nodos de C están en Q a la hora de que el algoritmo se regrese a la llamada de $Tarjan(n_i)$. Dado que el primer nodo en C empujado a la pila fue n_i, entonces en el while loop sacaremos todos los nodos de C necesariamente hasta extraer n_i. Por último, no puede ocurrir que un nodo que no esta en C sea extraído durante el while loop de $Tarjan(n_i)$, ya que formaría parte de otro circuito distinto a C y hubiese sido extraído por el while loop del representante de su circuito.

Este mismo algoritmo por etiquetamientos puede ser utilizado para detectar puntos de articulación y puentes en un grafo no dirigido con ligeros cambios, intente demostrarlo. Por último

analizaremos el tiempo de complejidad del algoritmo

Proposición 4.5.3

Dado un grafo dirigido $G = (V, E)$ el algoritmo de Tarjan toma tiempo lineal en $|V| + |E|$

:

Este algoritmo es un simple DFS modificado con la particularidad de extraer nodos de una pila Q que de hecho esto puede ser implementado con arreglos para saber si algun nodo está en Q o no. Además cada nodo es insertado y extraído de Q exactamente 1 vez, por lo que el tiempo total de $Tarjan$ *sería* $\theta(|V| + |E|)$

4.6 2-SAT

Una formula normal conjuntiva, o CNF, es una fórmula logica en la que solo existen conjunciones entre clausulas. Una **clausula** es una disjunción de varias variables que toman valor *true* o *false*. Veamos el siguiente ejemplo

$$(x \vee y \vee z) \wedge (\bar{x} \vee b \vee c) \wedge (c \vee d \vee e)$$

En este caso vemos que cada clausula posee tres *literales*, es decir la aparicion de una variable o su negada. En este caso la instancia es una fórmula expresada como 3-CNF ya que cada clausula posee maáximo 3 literales. El problema es encontrar una solución a la fórmula es decir una asignacion a las variables tal que la fórmula evalue a verdadero. En general este problema es muy difícil de resolver, y hasta el día de hoy no se conoce un algoritmo en tiempo polinomial capaz de resolverlo de hecho este problema es **NP-completo**. Sin embargo, en el caso en el que cada clausula esta acotada a tener máximo dos literales sí que es posible y de hecho ya tenemos las herramientas para resolverlo.

Vamos a recurrir a lógica simbólica, recordando que $a \Rightarrow b \equiv \neg a \vee b$. Esto quiere decir que por cada disjunción (la cual posee máximo dos literales) tenemos dos implicaciones equivalentes: dado $a \vee b$ las dos implicaciones son $\neg a \Rightarrow b$ y $\neg b \Rightarrow a$.

Ahora la pregunta es cómo saber si es imposible generar alguna asignacion a las variables que haga que la fórmula evalue a verdadero. Una forma de asegurar que no es posbile es mediante la propiedad transitiva de las implicaciones, donde sabemos que

$$a \Rightarrow b \wedge b \Rightarrow c \equiv a \Rightarrow c$$

Por otro lado, si generamos todas las implicaciones de la fórmula que nos dan en el problema y resulta que en algun punto para alguna variable x ocurre que $x \Rightarrow \bar{x}$ y además $\bar{x} \Rightarrow x$ sabemos que esto solo significa que $x \equiv \bar{x}$, es decir ocurre una contradicción. La pregunta que nos podemos hacer es si esta condición es necesaria y suficiente para saber que no existe una asignación válida de las variables (que dé *true* al sustituirlas) y para esto tenemos la siguiente proposición.

Proposición 4.6.1

Dada una fórmula en 2-CNF, si sustituimos cada disjunción por dos implicaciones equivalentes a la disjunción entonces existe una asignación válida a la fórmula original $\iff$ el conjuntos de implicaciones no llevan a una contradicción

:

- *($\Leftarrow$) Por reducción al absurdo, digamos que el conjunto de implicaciones llevan a una contradicción de la forma $x \equiv \bar{x}$. Veamos entonces que no puede haber una asignación válida que satisfaga la fórmula de 2-CNF, esto se debe a que no hicimos nada más que manipular la fórmula en 2-CNF agregando implicaciones a ella lo cual es equivalente a la fórmula original. Más aún, llegar a una conclusión de la forma $x \equiv \bar{x}$ y agregarla a la instancia del problema, daría un problema de satisfasibilidad equivalente al problema original (nuevamente) que concluye por no tener solución ya que ya que de haberla entonces en particular evaluaría como verdadero $x \equiv \bar{x}$, pero esto es falso para cualquier variable booleana.*

- *($\Rightarrow$) Supongamos ahora que existe una asignación válida a la fórmula original. Por el mismo argumento de la primera parte, la transformación de la fórmula a imlpicaciones genera una fórmula equivalente, esto quiere decir que de haber una contradicción en particular la asignación válida que estamos asumiendo que existe estaría pudiendo evaluar a verdadero una contradicción.* ■

Ahora bien, la pregunta es como podemos generar un algoritmo que nos diga si una fórmula en $2-CNF$ es posible de satisfacer o no. Por último se deja como ejercicio como modificar el algoritmo que veremos para devolver una asignación de las variables que evalúe a verdadero la fórmula.

Primero que nada vamos a modelar el problema como un grafo dirigido, donde cada literal representa un nodo. La siguiente pregunta es saber que tipo de arcos existen entre los nodos, para esto convertimos cada implicación de la manera $a \Rightarrow b$ en un arco (a, b) que pertenece al grafo. La manera de saber si existe una contradicción en la instancia dada es mediante el mismo grafo, estamos interesados en saber si dada una variable sus dos literales pertenecen al mismo circuito del grafo construido, de ser así entonces una contradicción existe por los argumentos de los párrafos anteriores, entonces acabamos de reducir 2-CNF a un problema de encontrar los circuitos de un grafo.

5

SECCIÓN

Caminos más cortos: caminos de costo mínimo

Hasta ahora hemos visto grafos donde cada arco tiene un costo de 1, es decir cada arco es igual de costoso de recorrer, sin embargo en la vida real cuando queremos viajar entre dos ciudades (nodos) es posible que existan dos autopistas distintas que las conecten (arcos) que recorrerlas tomen una cantidad de tiempo diferente. Este es el problema al que queremos enfrentarnos ahora, más aún estamos asumiendo que cada arco tiene un coso no negativo, sin emgargo es posible de asumir que algunos tendrán costos negativos.

5.1 Introducción a caminos de costo mínimo

Dado un grafo $G = (V, E)$ ahora queremos asociar una funcino de costos a los arcos

$$c : E \to \mathbb{R}$$

Ahora lo siguiente es definir cual sería nuestro problema a resolver: dado G, un grafo cualquiera (dirigido o no), deseamos poder calcular un camino de costo mínimo desde un nodo fuente $u \in V$ a un nodo objetivo $v \in V$ tal que el costo de ese camino sea mínimo, es decir

Definición 5.1.1: Camino de costo mínimo entre dos nodos

Es un camino tal que no exista otro camino de menor costo entre un par de nodos, esto lo podemos denotar como

$$\min_{p \in P_{u,v}} c(p) = \min_{p \in P_{u,v}} \sum_{e \in p} c(e)$$

Donde $P_{u,v}$ es el conjunto de todos los posibles caminos en G desde u hasta v. Además denotamos como el costo mínimo de un camino entre u y v como

$$\delta(u,v)$$

Donde $\delta(u,v) = \infty$ si no existe un camino de costo mínimo entre u y v.

Con está definición entonces nos preguntamos cómo es posible que no exista un camino de costo mínimo. La respuesta: es posible tener arcos de costo negativo y con esto es posible crear un ciclo cuyo costo sea negativo. Luego si encontramos un camino que pase por un ciclo de costo negativo entre un par de nodos, entonces recorrer mas veces el ciclo de costo negativo se traduce a un camino de menor costo, por lo que no existiría un mínimo. Vease la figura ??

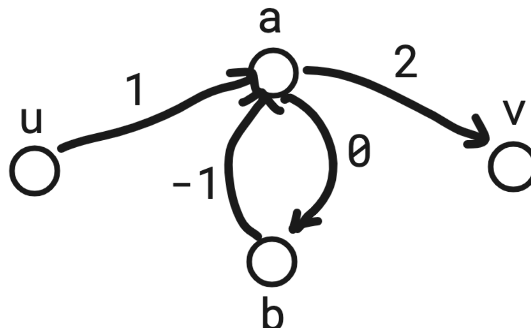


Figure 11: Grafo sin un camino de costo mínimo entre u y v, pasar mas de una vez entre a y b reduce cada vez más el costo final del camino

Ahora, asumir que existen lados de costos negativos conducen a la posible existencia de ciclos de costo negativo. En futuras secciones veremos como podemos resolver problemas en el caso de la existencia o inexistencia de ellos y podremos observar la diferencia de complejidad de ambos algoritmos.

Por otro lado, estaremos especialmente interesados en los algoritmos de camínos mínimos desde una sola fuente, es decir dado un nodo cualquiera $u \in V$ de un grafo $G = (V, E)$ nos interesa poder

encontrar el tamaño de todos los caminos mínimos entre u y todo $v \in V$. Para este tipo de problema definimos entonces $\delta_u(v)$ como el camino de tamaño mínimo entre u y v para todo $v \in V$. Ahora intenemos resolver con una idea muy sencilla el problema de caminos de costo mínimo con una sola fuente, nuestra idea es la siguiente:

- Inicialmente asignamos a cada $v \in V$ un valor que sería $d[v] = \infty$
- El valor $d[v]$ representa el costo del mejor camino hallado hasta el momento
- Inicialmente podemos asignar a la fuente $u \in V$ el valor de $d[u] = 0$
- luego iremos decresiendo $d[v]$ para algún $v \in V$ de alguna manera. Y queremos que siempre ocurra que $d[v] \geq \delta_u(v) \forall v \in V$
- iteramos sobre el punto anterior

Ahora procederemos a ver el algoritmo

Algorithm 4: Algoritmo de caminos de costo mínimo con una sola fuente

```
Input:
    u ∈ V
    G = (V, E)
d[v] ← ∞ ∀v ∈ V
d[u] ← 0
while ∃e = (x, y) ∈ E : d[y] > d[x] + c((x, y)) do
    e = (x, y) ← Elegir e ∈ E que cumpla con la condición del while
    d[y] ← d[u] + c(e)
```

Al proceso de actualizar la distancia d dado un arco a considerar lo llamamos *relajación* y lo usaremos en los próximas secciones. Ahora necesitamos saber si este algoritmo es correcto (si la solución que encuentra es la correcta para cualquier instancia) y el tiempo de complejidad.
Con un ejemplo podemos ver que el tiempo de nuestro algoritmo es al menos exponencial, lo cual no es ideal pero no esta mal para un primer enfoque. En la figura **??** si existieran n nodos los arcos que salen desde v_0 y v_1 tienen costo $2^{\frac{n-1}{2}}$, los de la siguiente tripleta costo $2^{\frac{n-2}{2}}$ y así sucesivamente. El nodo fuente es v_0 y en este caso, nuestro algoritmo pudo haber tomado inicialmente todos los arcos inferiores para calcular una primera aproximación de caminos de costo mínimo para cada nodo. Sin embargo, estos caminos no son óptimos e inicialmente $d[v_{n-1}] = 2 \times (2^{\frac{n-1}{2}} + ... + 1)$. Luego una manerad disminuir este número en una unidad es que se tome en cuenta el arco (v_{n-3}, v_{n-1}). Luego la siguiente manera de disminuir este número en una unidad es tomando primer en cuenta el arco (v_{n-5}, v_{n-3}) lo cual hace que se reduzca $d[v_{n-2}]$ en 2, luego tomando en cuanta el arco (v_{n-3}, v_{n-2}) y (v_{n-2}, v_{n-1}), se vuelve a reducri $d[v_{n-1}]$ en una unidad. Iterando este proceso, reduciriamos entonces $O(2^{\frac{n}{2}})$ veces $d[v_{n-1}]$ hasta llegar a su camino óptimo.
La manera de ver la correctitud del algoritmo es en realidad no de tanta importancia como la siguiente observación

Proposición 5.1.2: Subsestructura óptima de los caminos de costo mínimo

En un camino de costo mínimo, cualquier subcamino es también óptimo

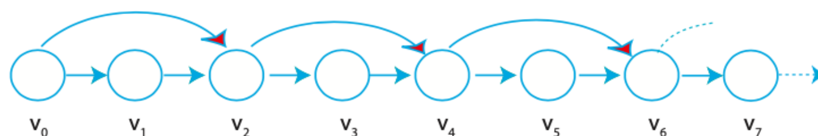


Figure 12: Instancia que genera tiempo exponencial en el algoritmo general de caminos de costo mínimo

: *Tomemos en cuenta un camino de costo mínimo cualquiera* $p = (v_0, v_1, ..., v_k)$. *Asumamos ahora que el camino entre dos nodos* v_i *y* v_j *con* $i < j$ *no es de costo mínimo sino que existe un mejor camino entre ellos que sería* $p' = (v_i, v'_0, v'1, ..., v_j)$. *Entonces podríamos reemplazar a* p *por* $p'' = (v_0, ..., v_i, v'_0, v'_1, ..., v_j, .., v_k)$ *y ocurre que* $c(p'') < c(p)$, *pero esto contradice la hipótesis de que* p *era un camino de costo mínimo entre* v_0 *y* v_k.

Por último, tenemos una propiedad que usaremos también en las próximas secciones

Proposición 5.1.3: Desigualdad triangular

Dado un grafo $G = (V, E)$ que no contiene ciclos de costo negativo tenemos que para todo $u, v, x \in V$ ocurre

$$\delta(u, v) \leq \delta(u, x) + \delta(x, v)$$

5.2 Camino de costo mínimo en un DAG

Para un grafo que posee un orden topológico, es decir donde no hay ciclos existe un algoritmo muy sencillo para resolver el problema de camino de costo mínimo, lo necesario a hacer es empezar desde el nodo o nodos raices y relajar cada arco que sale de ellos. Luego eliminamos los arcos que cabamos de relajar y continuamos procesando los nodos del arco en un proceso similar al proceso de computar un orden topológico. Este algoritmo toma tiempo lineal en el numero de nodos y arcos.

5.3 Dijkstra

Ahora supongamos que para cada arco $e \in E : c(e) > 0$, en el algoritmo de Dijkstra queremos mantener siempre un conjunto de nodos, S, al cual el camino mínimo ya ha sido calculado. Luego queremos seleccionar algun nodo nuevo a ese conjunto y luego relajar todos los arcos que salen del nodo nuevo. Más aun, este algoritmo necesita usar un priority queue para determinar el próximo nodo a procesar. Veamos el algoritmo

Algorithm 5: Algoritmo de Dijkstra

Input:

$u \in V$

$G = (V, E)$

$S \leftarrow \emptyset$

$Q \leftarrow V$ insertamos todos los nodos en un priority queue

$d[v] \leftarrow \infty \; \forall v \in V$

$d[u] \leftarrow 0$

while $Q \neq \emptyset$ **do**

  $v \leftarrow$ extraer nodo de mínimo d de Q

  $S \leftarrow S \cup \{v\}$

  for $e = (v, x) \in \delta^+(v)$ **do**

    if $d[x] > d[v] + c(e)$ **then**

      $d[x] \leftarrow d[v] + c(e)$ // Paso de relajación

Para entender el porqué este algoritmo funciona veamos entonces primero una porposición.

Proposición 5.3.1: El paso de relajación es seguro

Un paso de relajación mantiene el invariante $d[v] \geq \delta(u, v)$ para todo $v \in V$

: *Procedemos por inducción, en el número de veces que el paso de relajación haya sido usado. Inicialmente se cumple en ambos algoritmos que hemos visto hasta ahora. Ahora supongamos queremos relajar el arco* (x, y), *asumiendo que el nodo u es la raíz de la que queremos calcular todos los caminos mínimos sabemos entonces que hasta el momento tenemos que* $d[x] \geq \delta(u, x)$. *Luego por la desigualdad triangular, sabemos que* $\delta(u, y) \leq \delta(u, x) + \delta(x, y)$ *y esto quiere decir que* $\delta(u, y) \leq d[x] + c((x, y))$ *ya que* $d[x] \geq \delta(u, x)$ *y además el arco* (x, y) *tiene mayor o igual costo al camino mínimo entre* x *e* y. *Esto quiere decir que al asignar* $d[y] = d[x] + c((x, y))$ *mantenemos la invariante.* ■

Ahora nos preguntamos si el algoritmo de Dijsktra es correcto y cómo podemos probarlo. La idea para probar la correctitud es dandonos cuenta de que en el momento que un vertice $v \in V$ es agregado a S es porque $d[v] = \delta(u, v)$, si logramos probar esto significa que el algoritmo de Dijkstra está asignando en cada paso en realidad el camino mas corto.

Proposición 5.3.2: Caminos óptimos

Cada vez que Dijkstra agrega un nodo $v \in V$ a S implica que $d[v] = \delta(u, v)$ donde u es la raíz desde donde llamanos inicialmente al algoritmo

: *Procedemos por inducción en el costo mínimo de los caminos hacia los nodos del grafo $G = (V, E)$. Inicialmente $d[v] = \infty$ para todo nodo $v \in V$ y $d[u] = 0$, entonces u es el primer nodo extraído del priority queue y agregado a S. En este caso $d[u] = 0$ que es correcto ya que $\delta(u, u) = 0$ debido a la ausencia de arcos de costo negativo. Además por el mismo argumento ningún otro nodo en V puede ser alcanzado con un costo de 0, es decir que todos los nodos cuyo camino desde u tienen costo 0 han sido agregados a S.*

Ahora asumamos que todos los nodos en V que cumplen que $\delta(u, v) \leq q$ han sido agregados a S. Supongamos que un camino con costo mínimo que tiene un costo mayor a q tiene costo q' y asumamos que ese camino es $P = (v_0, e_0, ..., e_{k-1}, v_k)$ donde v_k no pudo haber sido agregado a S por hipótesis.

Luego, v_{k-1} ya existe en S de lo contrario P sin el último arco formaría un camino de menor costo que P y contradice nuestra hipótesis. Esto quiere decir que existió un paso de relajación desde v_{k-1} usando e_{k-1} que asignó $d[v_k] = d[v_{k-1}] + c(e_{k-1})$. Observemos que cuando esto octurre entonces $d[v_k] = \delta(u, v_k) = c(P)$, ya que $d[v_{k-1}] < q'$ por lo que por que en realidad $d[v_k] = \delta(u, v_{k-1}) + c(e_{k-1})$ que es exactamente el costo de P el camino de menor costo hasta v_k ■.

Proposición 5.3.3: Dijkstra es correcto

El algoritmo de Dijkstra computa $\delta(u, v) \forall v \in V$ donde u es la raíz desde donde llamanos inicialmente al algoritmo ■

Proposición 5.3.4

El tiempo de complejidad de Dijkstra es $\theta(|V|\log(|V|) + |E|\log(|V|))$ usando un heap binario

: *Veamos que existen:*

- $\theta(|V|)$ *inserciones al priority queue inicialmente*
- $\theta(|V|)$ *operaciones de extraer el mínimo elemento*
- $\theta(|E|)$ *operaciones de decrementar el valor de d para algun nodo*

Implementar este tipo de operaciones con un heap binario nos tomaria $\theta(\log(|V|))$ luego de haber insertado inicialmente los $|V|$ elementos. Esto quiere decir que nuestro tiempo total de complejidad sería $\theta(|V|\log(|V|) + |E|\log(|V|))$ ■

Con un heap de fibonacci el total de operaciones se reduce a $\theta(|V|\log(|V|) + |E|)$.

5.4 Bellmand-Ford: caminos cortos en grafos con arcos costo negativo

Anteriormente dijimos que si un grafo tiene un ciclo de costo negativo entonces no existiría un camino de costo mínimo, sin embargo, esto no quiere decir que no puedan existir en el grafo

arcos de costo negativo; nuestra única condición entonces es que no existan **ciclos** de costo negativo y si esto nunca ocurre entonces podremos encontrar caminos de costo mínimo para cada nodo del grafo.

Ahora enfoquemonos en el algorimo general para calcular caminos mínimos desde una sola fuente que vimos anterior mente (8), el problema del algoritmo era básicamente la elección del arco a relajar, y mostramos un ejemplo de una instancia que conlleva a una complejidad exponencial en el número de nodos. Sin embargo, es posible de mejorar este tiempo, pero primero definamos una subrutina de relajar un arco

---

Algorithm 6: Relajar

Input:
- $x \in V$
- $y \in V$
- Acceso a d

if $d[y] > d[u] + c(x, y)$ **then**
- $d[y] \leftarrow d[u] + c(x, y)$
- **return** *true*

return *false*

---

y ahora podemos introducir el algoritmo de Bellman-Ford

Algorithm 7: Bellmand-Ford

Input:
- $u \in V$
- $G = (V, E)$

$d[v] \leftarrow \infty \ \forall v \in V$

$d[u] \leftarrow 0$

for $i = 1$ *to* $|V| - 1$ **do**
- **for** $e = (x, y) \in E$ **do**
  - Relajar(x,y)

for $e = (x, y) \in E$ **do**
- **if** *Relajar(x,y)* **then**
  - Reportar ciclo negativo

Ahora lo que debemos demostrar es que este algoritmo computa el óptimo (el camino mínimo para cada nodo del grafo) o reporta un ciclo negativo en su defecto.

Proposición 5.4.1: Bellman-Ford es óptimo

Si un grafo $G = (V, E)$ no contiene ciclo negativos entonces después de que Bellman-Ford termine se tiene que $d[v] = \delta(u, v)$ para todo nodo $v \in V$

: *Para demostrar este teorema tomemos en cuenta un camino de costo mínimo cualquiera $P = \langle v_0, v_1, ..., v_k \rangle$ donde $v_0 = u$ es la raíz de donde se computan los caminos mínimos. Nuestra primera observación es que al no haber ciclos de costo negativo entonces P tiene que ser un camino simple por lo tanto $k < |V|$. La siguiente observación es que inicialmente $d[v_0] = \delta(u, v_0) = \delta(u, u) = 0$. Luego de la primera iteración, entonces se calcularía $d[v_1] = \delta(u, v_1) = \delta(u, v_0) + c(v_0 v_1)$ gracias a la relajación del arco entre v_0 y v_1 y sabemos que este es el óptimo gracias a la subestructura óptima de los caminos de costo mínimo (teorema demostrado anteriormente). Lo mismo ocurre para cada iteración siguiente hasta la $k-1$ hasta calcular entonces el costo óptimo para cada nodo en P. Por lo tanto, si iteramos $|V|-1$ veces calcularíamos para cualquier camino mínimo P' en G el costo correcto para cada nodo del grafo.* ■

Proposición 5.4.2: Bellman-Ford reporta la existencia de ciclos negativos

Si después de $|V|-1$ iteraciones Bellman-Ford falla en asignar $d[v] = \delta(u, v)$ para un nodo $v \in V$ desde un nodo fuente u es porque existe un ciclo de costo negativo en el grafo a considerar

: *Dada la existencia de algún ciclo negativo en un grafo, después de $|V|-1$ iteraciones de Bellman-Ford tiene que ocurrir que alguno de los nodos del ciclo negativo puede ser relajado, ya que un ciclo negativo implica la existencia de un camino de costo mínimo de tamaño mayor igual a $|V|$ por lo que el algoritmo caería en el reporte del ciclo negativo.* ■

5.5 Caminos simples de costo máximo y mínimo

Ambos problemas de encontrar el camino simple de costo mínimo y máximo en un grafo cualquiera son problemas $NP-hard$ para los que no se conoce solución en tiempo polinomial. Fijémonos que si negaramos el costo de los arcos y usaramos Bellman-Ford para intentar calcular el camino simple de costo máximo el algoritmo fallaría reportando ciclos negativo para aquellos grafos que poseen algun tipo de ciclo. El problema análogo de computar el camino simple de costo mínimo es también NP-Hard por lo que las esperanzas de encontrar nu algoritmo polinomial para ello son bastante bajas.

5.6 Caminos de costo mínimo entre cada par de vertices

Ahora supongamos que tenemos un grafo en el que no existen ciclos de costo negativo y quisieramos calcular el camino de costo mínimo entre cada par de vertices. Una forma de hacer esto es obviamente usando los algoritmos vistos anteriormente, es decir podríamos ejecutar Dijkstra $|V|$ veces, una vez desde cada nodo del grafo lo cual nos tomaría un tiempo de ejecución entonces de $\theta(|V|(|V|\log(|V|) + |E|\log(|V|)))$. Nos preguntamos ahora si es posible intentar obviar el término $|E|$ cuando $|E|\ |V|^2$, por fortuna este algoritmo ya existe y esta basado en una técnica parecida a programación dinámica que veremos en un futuro.

Imaginémonos ahora que tenemos un algoritmo que en cada iteración encuentra los caminos mínimos entre cada par de vertices que usa hasta únicamente los primeros j vertices del grafo

donde j es el número de la iteración. Esto tiene como caso base los caminos de largo 0 de cada nodo hasta sí mismos. Asumismos un orden arbitrario de los nodos del grafo $n_1, n_2, ..., n_{|V|} \in V$

Lo siguiente que podríamos hacer en la iteración j es entonces intentar unir dos caminos cuyo nodos intermedios sean unicamente del conjunto de los primeros $j-1$ nodos de consideramos, es decir, dado dos nodos $u, v \in V$ a los que intentamos calcular el camino de costo mínimo usando los primeros j nodos del grafo donde la única diferencia es que ahora el nodo j podría ser un intermediario y ayudar a construir un mejor camino entre u y v. Esto nos daría una recursión de la forma

$$c(u, v) = \min_{w \in V}(c(u, n_j) + c(n_j, v))$$

donde $c(x, y)$ indica el costo de un camino mínimo extre $x, y \in V$ en alguna iteración de nuestro algoritmo. Por último, puede suceder que el mejor camino entre $u, v \in V$ ya haya sido calculado ópotimamente en la iteración anterior, que es un caso que hay que cuidar por separado. Tomando en cuenta esta observación tenemos que en cada iteración tendríamos que calcular entonces

$$c(u, v) = \min \begin{cases} \min(c(u, n_j) + c(n_j, v)) \\ c(u, v) \end{cases}$$

Con esto llegamos entonces al algoritmo de Floyd-Warhsall

Algorithm 8: Floyd-Warshall primera ver

Input:

$G = (V = \{n_1, n_2, ..., n_{|V|}\}, E)$

$d[x, y] \leftarrow \infty \; \forall x, y \in V$

$d[x, x] = 0 \; \forall x \in V$

$d[x, y] = c((x, y)) \; \forall (x, y) \in E$

for $j = 0$ *to* $|V| - 1$ **do**

  for $u, v \in V$ **do**

    $d[u, v] = \min \begin{cases} d[u, n_j] + d[n_j, v] \\ d[u, v] \end{cases}$

Lo más fácil de analizar en este algoritmo es su complejidad, luego analizaremos su correctitud.

Proposición 5.6.1

El tiempo de complejidad de Floyd Warshall es $\theta(|V|^3)$

: *Uniendo ambos For loops del código tenemos que la actualización de las distancias que ocurre en la aprte mas interna del algoritmo se ejecutarían un total de $\theta(|V|^3)$ veces. Por otro lado, la actualización de la matriz de distancia toma cada vez tiempo constante si representamos a d como una matriz bidimensional, por lo tanto esto nos da el tiempo de comlpejidad propuesto.*

Proposición 5.6.2

El algoritmo de Floyd-Warshall es correcto

: *Procedemos por inducción en el número de iteraciones sobre el foor loop mas externo. Antes de la iteración 0, vems que* $d[x,y]$ *para* $x,y \in V$ *es el costo del mejor camino entre* x *e* y *sin ningun tipo de nodos intermediarios ya que* $d[x,x] = 0$ *para todo* $x \in V$ *y el resto de la tabla de distancias* d *es simplemente seteado a* ∞.
Ahora asumamos antes de la iteración k *tenemos que* $d[x,y]$ *contiene el costo del camino mínimo entre* x *e* y *usando sólamente los primeros* $k-1$ *nodos. Lo siguiente que nuetro algoritmo va a hacer es tratar de unir cualquiera dos caminos pasando por el nodo* k, *dado* $x,y \in V$ *un camino mínimo entre ellos podría incluir a* n_k, *de ser este el caso entonces el costo del camino mínimo usando los primeros* k *nodos será calculado ya que tomamos en cuenta un camino de costo mínimo desde* x *hasta* n_k *conteniendo los primeros* $k-1$ *nodos y hasta* y *desde* n_k *también conteniendo los primeros* $k-1$ *nodos. Por inducción, sea el camino* $P = \langle x, \ldots, n_k \rangle$ *aquel cuyo costo es* $c(P) = d[x, n_k]$ *hasta el momento de la iteración* k *y de la misma manera el camino* $P' = \langle n_k, \ldots, y \rangle$, *ambos son caminos de costo mínimo usando los primeros* $k-1$ *nodos del grafo. Por lo tanto, unirlos sería un camino de costo mínimo entre* x *e* y *usando los primeros* k *nodos del grafo (a menos que* n_k *no ayude a mejorar el costo ya calculado para un camíno de costo mínimo entre* x *e* y*). Luego de la iteración* $|V|-1$ *entonces la tabla de distancias tendra el calculo del costo del camino de costo mínimo para cada par de vertices usando los |V| nodos del grafo, es decir los caminos de costo mínimos del grafo*■

SECCIÓN 6

Búsqueda informada

Anteriormente vimos como podemos encontrar caminos de costo mínimo en grafos sin ciclos de costo negativo, ahora quisieramos hacer lo mismo pero con un algoritmo que es dependiente del dominio, es decir, del tipo de problema.

Supongamos que tenemos un tipo de problema cuyo dominio conocemos un poco, digamos el siguiente juego de deslizar piezas hasta ordenarlas

1	2	3
	4	6
7	5	8

Figure 13: Sliding puzzle

Lo primero que tendríamos que preguntarnos es como representar este problema como un

problema de grafos. Lo primero es que cada nodo o estado lo podemos representar como alguno de las posibles configuraciones del juego, que no son fáciles de definir ya que no toda configuración del puzzle es válida pero es al menos facil de describir. Luego podemos optar por la representación implicita, donde decimos que dos nodos estan unidos por un arco si es posible ir de una configuación a otra mediante un solo paso, donde un paso es simplemente un movimiento de una pieza. Entonces lo que tendríamos es un grafo de la siguiente forma

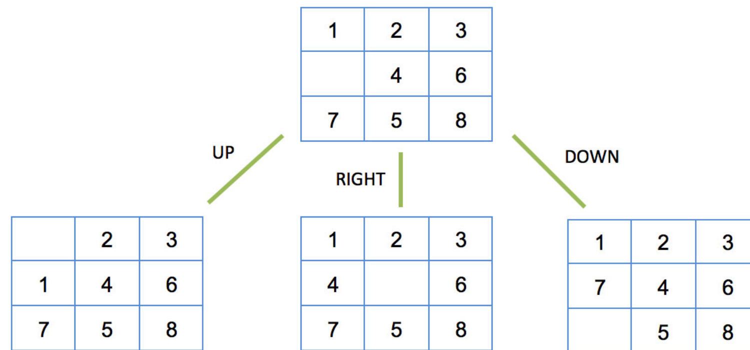


Figure 14: Movimientos en un sliding puzzle

Algunos nodos tendras 4 arcos conectados a ellos (el que tiene el hueco en el medio) y otros tendras 3 o 2 arcos incidentes. Por otro lado, este grafo es un grafo no dirigido, ya que es posible ir de un nodo a otro y regresar por el mismo movimiento pero haciendolo inversamente.

6.1 Algoritmo A^*

A la hora de resolver un problema, como persona podemos tener un presentimiento de que tan lejos o cerca estamos de una solución, por ejemplo cuando hace falta un ultimo movimiento para resolver el problema. También podríamos tener una idea de que tan malo es un nodo en el sentido de qué tan lejos está un nodo del nodo al que quisieramos llegar, en nuestro caso la solución del sliding puzzle.

En el algoritmo que veremos a continuación, a cada nodo asociamos una función del costo probable entre ese nodo y el nodo objetivo. Este costo probable que es una función $f(x)$ para un nodo $x \in V$ de nuestro grafo $G = (V, E)$ donde $f(x) = g(x) + h(x)$, siendo $g(x)$ el mejor costo real hasta ahora encontrado y recorrido para alcanzar x y $h(x)$ es nuestra estimación de cuanto faltaría para llegar al nodo objetivo $t \in V$, la función $h : V \to \mathbb{R}$ es una ***heuristica***. Ahora bien nuestra estimación podría sobreestimar el costo real hasta llegar a t (mediante un camino óptimo) o lo podría subestimar.

Nuestro problema sigue siendo el problema de encontrar el camino mínimo o de menor costo hasta llegar a un nodo objetivo. Ahora que tenemos asociado a cada nodo del grafo un costo real y un costo estimado hasta llegar al nodo objetivo lo siguiente es entender qué podriamos hacer con ello. Procedamos a ver entonces el siguiente algoritmo

Algorithm 9: A^* (Peter Hart, Nils Nilsson and Bertram Raphael; 1968)

Input:
  $u, t \in V$
  $G = (V, E)$

$S \leftarrow \emptyset$ // Lista de nodos abiertos
$C \leftarrow \emptyset$ // Lista de nodos cerrados
Insertar u a S
$u.f \leftarrow 0$
while $S \neq \emptyset$ **do**
  $x \leftarrow S$ // siendo x el nodo en S con menor $x.f$
  $S \leftarrow S \setminus \{x\}$
  if $x = t$ **then**
    Parar la búsqueda y retornar $x.f$ como el costo para alcanzar t
  for $\{x, y\} \in \delta(x)$ **do**
    Computar $y.h$
    if $y \in S$ *con menor* $y.f$ **then**
      Continuar con siguiente arco en $\delta(x)$
    if $y \in C$ *con menos* $y.f$ **then**
      Continuar con siguiente arco en $\delta(x)$
    Remover y de S y de C si existe
    Insertar y en S
  Insertar x en S

A primera vista parece un algoritmo bastante complicado pero la verdad es que este algoritmo, es el mismo algoritmo de Dijkstra con 2 modificaciones:

- La función de costo ya no es el camino hasta cada nodo sino ese mismo valor aunado a una función de costo probable para llegar a un nodo objetivo
- Está permitido reabrir nodos, es decir, podemos visitar un mismo nodo mas de una vez al mejorar su costo real + costo estimado hasta un nodo objetivo.

La pregunta ahora sería si al ejecutar A^* ocurre que cuando el algoritmo para es porque ha encontrado un camino de costo óptimo. Antes de intentar llegar a esta conclusión veamos primero esta definición

Definición 6.1.1: Algoritmo admisible

Un algoritmo es ***admisible*** si encuentra el camino óptimo desde un nodo inicial a uno final en un grafo dado.

6.2 Admisibilidad de A^*

Para probar la admisibilidad de A^* bajo ciertas condiciones nos vamos a basar en un lema, un corolario y luego probaremos su admisibilidad.

Lema 2

Dado un grafo $G = (V, E)$, para cada nodo $n \in V$ no cerrado por A^* para cada camino P desde un nodo inicial s hasta n existe algun nodo abierto $n' \in V$ que pertenece a P donde $g(n')$, o $n'.g$, es el costo un camino óptimo hasta n'

: *Sea* $P = \langle s = n_0, n_1, ..., n_k = n \rangle$ *un camino óptimo hasta llegar desde* s *a* n. *Si* n *está abierto, entonces es trivial, ninguna iteración del algoritmo ha sido ejecutada aún por lo que* $s.f = 0$ *es el camino óptimo hasta* s *y el lema es cierto.*

Ahora Supongamos que s *esta cerrado, digamos que* Δ *es el conjunto de todos los nodos cerrados en* $x \in P$ *para los cuales* $x.g$ *es el valor del costo óptimo hasta* x. *Sabemos que* Δ *no es vacío ya que por lo menos* $s \in \Delta$. *Ahora llamemos* $n^* \in \Delta$ *el elemento en* P *que pertecene a* Δ *de mayor indice y llamemos* n' *su sucesor en* P *que podría bien ser* n. *Llamemos* $opt(x)$ *el valor del camino de costo mínimo desde* s *hasta* x *para cualquier* $x \in V$. *Tenemos que* $n'.g \leq n^*.g + c(n^*, n')$ *y además debido a que* $n^* \in \delta$ *tenemos que* $n^*.g = opt(n^*)$ *y por lo tanto* $n'.g \leq opt(n^*) + c(n^*, n')$. *Pero debido a que* P *es un camino óptimo, por la subóptimalidad de los caminos tenemos entonces que* $n'.g = g(n') = opt(n')$ *y además* n' *es abierto, de lo contrario pertenecería a* Δ ■

Con esto pasamos a siguiente corolario

Corolario 1

Dado un grafo $G = (V, E)$, si para cada nodo $x \in V$ ocurre que $h(x)$ subestima el costo real desde x hasta un nodo final $t \in V$ y A^* aún no ha terminado entonces para cada camino óptimo P desde el nodo de comienzo s hasta el nodo final t existe un nodo abierto $n' \in P$ tal que $f(n') \leq opt(t)$

:

Por el lema anterior, existirá un nodo abierto $n' \in P$ *con* $n'.g = g(n') = opt(n')$, *luego por definición de* f

$$f(n') = g(n') + h(n') = opt(n') + h(n') \leq opt(t) \blacksquare$$

Con esto llegamos a nuestro teorema deseado para probar que A^* si funciona

Teorema 6.2.1: *Dado un grafo* $G = (V, E)$, *si* h *es una heuristica admisible, es decir que* $\forall x \in V : h(x)$ *es menor o igual al costo de un camino óptimo desde* x *a un nodo final* t *entonces* A^* *es admisible*

: *Procederemos por reducción al absurdo, suponiendo que A^* no termina encontrando el camino óptimo hasta t. De ser este el caso, existen 3 posibles casos*

- *El algoritmo termina en un nodo no final: esto contradice la condición de terminación de A^*, por lo tanto no es posible.*
- *El algoritmo no termina: sea t un nodo final cualquiera que es alcanzable en una cantidad finito de pasos desde s el nodo inicial de la búsqueda. Dado que cada arco tenga un costo de al menos δ, cada nodo $n \in V$ mas allá de un costo de $M = \frac{opt(t)}{\delta}$ pasos desde s tendremos que $f(n) \geq g(n) \geq opt(t)$. Claramente ningun nodo mas allá de M pasos desde s será expandido ya que por el lema anterior el algoritmo elegiría primero cualquier nodo abierto en uno de los caminos óptimos hasta t.*
  Entonces la no terminación del algoritmo solo puede ser causada por la reapertura de nodos a M pasos de distancia de s, sin embargo existe solo una cantidad finita de caminos hasta cualquier nodo a M pasos de distancia, por lo tanto el algoritmo terminaría en algun punto de "optimizar" todos esos nodos.
- *La terminación ocurre en un nodo final t pero calculando algún costo no óptimo hasta t: si A^* termina en t con un costo $\widehat{opt}(t) > opt(t)$, por el corolario anterior debería haber existido antes de la terminación algun nodo n' en un camino óptimo entre s y t con $f(n') < opt(t) < \widehat{opt}(t)$ por lo que A^* hubiese escogido a n' antes que t* ■

Con esto hemos probado entonces que bajo una heuristica admisible entonces A^* es admisible. La heuristica más fácil que nos podemos imaginar que es admisible sería la heuristica constante $h(x) = 0$ para todo $x \in V$ en un grafo. Sin embargo, usando esta heuristica estaríamos haciendo lo mismo que un Dijkstra y la idea de A^* es usar un poco de información para guiar mejor la búsqueda. En el caso que estabamos estudiando del sliding puzzle, una heuristica podría ser calcular la suma de las distancias manhattan de cada pieza hasta su posición final correspondiente, esto obviamente es una heuristica admisible ya que representaría el mejor de los casos al resolver el puzzle y es una heurística que funciona muy bien en la práctica.

SECCIÓN 7

Árboles

Hasta ahora nos hemos enfocado en grafos en general, sin embargo en esta sección nos vamos a estar enfocando en árboles y arborescencias, unos tipos de grafos que debido a sus estructuras y sus propiedades permiten resolver problemas de una manera sencilla en comparación a grafos mas generales como es el caso del problema del diametro de un grafo.

7.1 Arboles y arborescencias

Definición 7.1.1: Árbol

Un **árbol** es un grafo conexo que no tiene ciclos

Proposición 7.1.2

Si un árbol tiene n nodos, entonces el árbol tiene $n-1$ arcos.

Definición 7.1.3: Árborescencia

Una **árborescencia** es un grafo en el que cada componente conexa es un árbol

Proposición 7.1.4

Si una arborescencia tiene n nodos y k componentes conexas, entonces la arborescencia tiene $n-k$ arcos.

7.2 Arbol cobertor de costo mínimo (Minimum Spanning Tree)

Definición 7.2.1: Árbol cobertor

Supongamos ahora que tenemos un grafo no dirigido $G(V,E)$ y queremos calcular un subgrafo $T = (V,E')$ con $E' \subseteq E$ tal que T es un grafo conexo y $|E'| = |V| - 1$, es decir T es un árbol. Además para cada elemento en V existe un arco en E' que lo hace conectarse al árbol. T es entonces un **árbol cobertor** en otras palabras un subárbol del árbol original.

Ahora asumamos que tenemos una función $w : E \to \mathbb{N}$ que indica el peso (weight) de un arco en un grafo dado, podemos entonces proceder a definir el costo de un subconjunto de E

Definición 7.2.2: Costo de un conjunto de arcos

Dado un grafo $G = (V,E)$ el costo o peso total de un conjunto E' es simplemente

$$w(E') = \sum_{e \in E'} w(e)$$

Definición 7.2.3: Árbol mínimo cobertor

Dado un grafo $G = (V,E)$ un árbol mínimo cobertor $T \subseteq E$ es un árbol cobertor tal que $w(T) \leq w(T')$ para todo árbol cobertor T' de G.

Nuestro objetivo ahora sería entonces calcular un árbol mínimo cobertor para un grafo dado como input. Para esto existen dos algoritmos bien conocicos: el algoritmo de Kruskal y el de Prim. Ambos algoritmos comienzan teniendo un árbol cobertor vacío e iteran buscando la siguiente "mejor arista" a agregar en el árbol cobertor para que sea también mínimo e iteran hasta que el conjunto de arcos mínimos acumulados forme un árbol. Antes de proceder queremos primero probar una de las propiedades que ayudan a comprender mejor estos algoritmos.

Definición 7.2.4: Corte de un grafo

Un **corte** de un grafo no dirigido $G = (V,E)$ es una partición $(S, V \setminus S)$ de sus vertices.

Una arista $\{u,v\} \in E$ **cruza el corte** $(S, V \setminus S) \iff u \in S \land v \in V \setminus S$

Un corte $(S, V \setminus S)$ **respeta** un conjunto de aristas $A \subseteq E$ si ningún arco de A cruza el corte.

Una arista que cruza el corte es **liviana** si cruza un corte del grafo y su peso es mínimo entre todas las aristas que cruzan el corte.

Esto es suficiente entonces para poder enunciar el siguiente teorema

Teorema 7.2.5: *Dado un grafo $G = (V,E)$ conectado y no dirigido con pesos en sus aristas y dado $A \subseteq E$ contenidas en algún MST de G. Sea ahora $(S, V \setminus S)$ un corte de que respeta a A, si tenemos una arista liviana $e \in E$ que cruza el corte $(S, V \setminus S)$ entonces $A \cup \{e\}$ esta contenido en algun MST de G*

:

Asumimos ahora que $A \subseteq E$ esta contenido en un MST $T = (V, E')$ de G. Tenemos que encontrar ahora un MST $T' = (V, E'')$ tal que $A \cup \{e\} \subseteq E''$ donde $e \in E$ es una arista liviana. Veamos que si $A \cup \{e\} \subseteq E'$ entonces no hay nada que demostrar ya que T sería el MST buscado.
Por otro lado $A \cup \{e\} \not\subseteq E'$, es decir que agregar e a T formaría un ciclo y ese ciclo contiene una arista $e' \in E'$ que cruza a $(S, V \setminus S)$. Consideremos ahora a $E'' = (E' \cup \{e\}) \setminus \{e'\}$, queremos ver que $T' = (V, E'')$ es un MST y $A \cup \{e\} \subseteq E''$.
Primero que nada T' es un árbol ya que T era un árbol y cuando quitamos e' y agregamos e en T sigue existiendo un camino para cada par de vertices en T'. Además se cumple que $w(T') = w(T) - w(e') + w(e) \leq w(T)$, ya que ambos e y e' cruzan el corte pero e tiene menor costo por ser una arista liviana y dado que $w(T') \leq w(T)$ y que T es un MST esto implica que T' es también in MST. Por último veamos que $e' \not\in A$ ya que cruza el corte con lo cual $A \cup \{e\} \subseteq E''$ ■

Corolario 2

Dado un grafo $G = (V, E)$ conectado y no dirigido con pesos en sus aristas y dado $A \subseteq E$ contenida en algún MST de G. Sea ahora el grafo $G_C = (V_C, E_C)$ el grafo de componentes de la arborescencia $G_A = (V, A)$, si existe una arista liviana $e = \{u, v\} \in E$ que conecta a componentes distintas de G_C, entonces $A \cup \{e\}$ esta contenido en algún MST de G

:

Dado dos componentes conexas C_1, C_2 de G_A tal que $u \in C_1 \wedge v \in C_2$, si consideramos el corte $(C_1, V \setminus C_1)$, vemos que ninguna arista de A cruza el corte, pero e si lo hace y además es una arista liviana por lo que por teorema $A \cup \{e\}$ sería un subconjunto de algun MST de G ■

Con esto ya tenemos las herramientas necesarias para presentar los algoritmos de Prim y Kruskal.

7.2.1 Algoritmo de Kruskal

El algoritmo de Kruskal comienza con cada nodo aislado siendo una componente conexa, luego mediante alguna arista liviana que conecte a dos componentes conexas puede ser agregar sin violar la invariante de generar un subconjunto de un MST.

Algorithm 10: Kruskal

Input:
$G = (V, E)$
$A \leftarrow \emptyset$
for $v \in V$ **do**
  hacer set para v
Ordenar las aristas en E de menor a mayor peso
for $\{u, v\} \in E$ *en orden de peso* **do**
  if $find(u) \neq find(v)$ **then**
    $A \leftarrow A \cup \{\{u, v\}\}$
    $union(u, v)$
return A

Vemos que cada vez que el algoritmo hace un union es porque los dos nodos del lado no perteneces

al mismo conjunto, es decir que la arista $\{u, v\}$ une dos componentes distintas y por lo tanto es una arista liviana que por el corolario anterior prueba la correctitud del algoritmo.

Por otro lado, queda discutir cual es la complejidad del algoritmo. Tomando en cuenta que cada set puede ser implementado con la implementación de disjoint set union (DSU) vista en algoritmos 2 cuya funciones de find y union son quasi constante. Es decir que el algoritmo completo tomaría tiempo $O(|E|\log(|E|))$ por el ordenamiento más las operaciones de find y union que tomarían tiempo $O(\alpha \times |E|)$ donde α es la inversa de la función de ackerman que es casi constante en la práctica. Es decir el tiempo total es entonces $O(|E|\log(|E|))$

7.2.2 Algoritmo de Prim

El algoritmo de Prim comienza con un solo nodo en el MST a construir. Luego la idea es crecer el árbol con una arista liviana que une la componente conexa del árbol construido hasta el momento con el conjunto de nodos en el complemento. De esta manera por teorema sabemos que el MST construido sería ciertamente un MST.

Algorithm 11: Prim

```
Input:
    G = (V, E)
    raiz
for v ∈ V do
    key[v] = ∞ // llave que usaremos en el priority queue
    π[v] = nil // arreglo de padres
key[raiz] = 0
q ← ∅ // priority queue ordenado por key
for v ∈ V do
    q.insert(v)
while ¬q.empty() do
    u ← q.top()
    q.pop() for {u, v} ∈ δ(u) do
        if v ∈ q ∧ w(u, v) < key[v] then
            key[v] = w(u, v)
            //esto decrementa el key de v en q π[v] = u
return π
```

En el algoritmo π denota el conjunto de padres de cada nodo, al final del algoritmo el MST esta descrito mediante este arreglo π ya que es suficiente para describir el árbol MST, es decir cada nodo tiene su nodo padre asignado, suficiente para describir un árbol.
Por otro lado, el algoritmo de Prim mantiene 3 invariantes

- El conjunto de aristas que forman el MST es $A = \{\{v, \pi[v]\} : \forall v \in V \setminus (Q \cup \{raiz\})\}$
- Los vertices del MST son aquellos que no estan en la cola
- Para todo vertice v con $\pi[v] \neq nil$
  - $key[v] < \infty$

- Si $v \in Q$, $key[v]$ es el peso de una arista liviana que conecta a v con el MST computado hasta el momento. Con la ayuda del teorema anterior sabemos que el algoritmo es correcto.

Por último veamos el análisis de complejidad del algoritmo de Prim:

- El tiempo de inicialización es $O(|V| + |V|\log(|V|))$
- La cantidad de operaciones en la cola es $O(|V|\log(|V|) + |E|\log(|V|) = |E|\log(|V|)$ debido al for mas interno en el algoritmo y a la inicialización de la cola.
- El for interno toma $|E|$ iteraciones sumado entre todos los nodos.
- El tiempo total es entonces $O(|E|\log(|V|))$

7.3 Diámetro de un árbol

SECCIÓN

8 Algoritmos de flujo máximo

En este capítulo vamos a enfocarnos en un tipo de problema diferente llamado *flujo en redes* que tiene un uso tan práctico como el de los de caminos mínimos. Además es un problema muy estudiado en el área de optimizaciones combinatorias e in vestigación de operaciones. Un flujo en una red puede ser visto como un flujo en una tubería de un edificio, donde éxiste una fuente de donde sale el flujo (el proveedor del servicio del agua) y un sumidero (nuestro apartamento) y la idea es encontrar cual es la cantidad máxima de flujo que puede ir al mismo tiempo desde la fuente al sumidero tomando en cuenta las limitaciones de las tuberías (cuanta agua puede pasar al mismo tiempo por ellas).

8.1 Flujos y flujo máximo

Supongamos ahora que tenemos un grafo dirigido $G = (V, E)$ con dos vertices $s, t \in V$ que son la fuente y el sumidero respectivamente. Además, los arcos de G tienen una capacidad asociada $c : V \times V \to \mathbb{R}^{\geq 0}$ que indica cuanto flujo puede pasar por el arco en un instante. Por lo general el grafo G está conectado y los únicos vertices fuente y sumidero son s y t respectivamente.
Además podemos asumir las siguientes suposiciones sin perder generalidad:

- Si E con tiene la arista (u, v), entonces $(v, u) \notin E$. De existir (v, u) en su lugar podemos introducir un nodo extra w y dos arcos (v, w) y (w, u) cuyas capacidades son igual a $c((v, u))$ en el grafo original. El grafo resultante es equivalente en cuanto al problema de flujo al grafo original.
- Si $e \notin E$ entonces $c(e) = 0$
- No existen lazos, es decir arcos de la forma (u, u) con $u \in V$

- Todo vertice pertenece a algún camino desde s a t. En este caso entonces necesariamente $|E| > |V| - 1$

Definición 8.1.1: Función de flujo

Una **función de flujo** es una función $f : V \times V \to \mathbb{R}^{\geq 0}$ tal que f indica la cantidad de flujo que pasa por un arco en un instante. A nosotros solo nos interesan los flujos tales que si $f(u, v) > 0$ entonces $(u, v) \in E$

Definición 8.1.2: Flujo

Un **flujo** es una función de flujo que satisface dos restriciones

- **Restricciones de capacidad**: para todo $u, v \in V$

$$0 \leq f(u, v) \leq c(u, v)$$

- **Conservación de flujo**: para todo vertice $u \in V \setminus \{s, t\}$

$$\sum_v f(u, v) = \sum_v f(v, u)$$

Definición 8.1.3: Valor de un flujo

El **valor de un flujo** f es

$$|f| = \sum_v f(s, v) - \sum_v f(v, s)$$

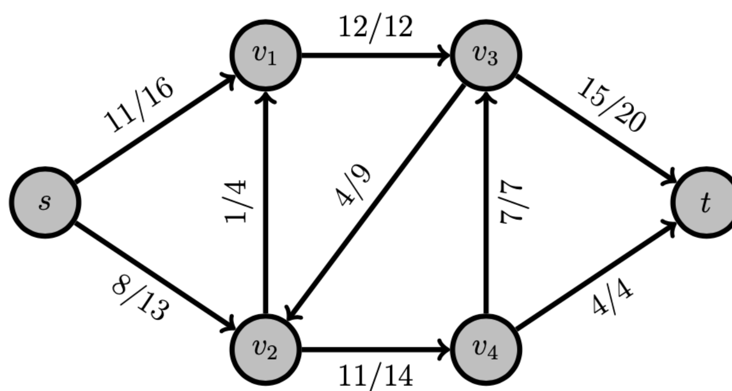


Figure 15: Ejemplo de flujo de valor 19. En el ejemplo los valores de los arcos son $f(u, v)/c(u, v)$ para el arco (u, v)

Algunos ejemplos de flujos son:

- Flujo nulo: $f_o(u, v) = 0$ para todo par de vertices

- Flujo a lo largo de un camino: Sea $p = (v_0, ..., v_k)$ un camino simple de s a t y sea $c(p) = \min\{c(v_{i-1}, v_i) : 1 \leq i \leq k\}$. Definimos el flujo a lo largo de p como

$$f_p(u,v) = \begin{cases} c(u,v) \text{ si}(u,v) \in p \\ 0 \text{ si}(u,v) \notin p \end{cases}$$

Indique por qué los ejemplos anteriores son flujos.

Definición 8.1.4: Problema de flujo máximo

Un flujo cuyo valor es mayor que cualquier otro flujo posible en un grafo es un flujo máximo. El **problema de flujo máximo** consiste en encontrar un flujo válido en un grafo con capacidades en sus arcos tal que sea máximo.

8.2 Algoritmo de Ford-Fulkerson

En general los algoritmos para resolver el problema de flujo máximo están basados en tres ideas

- Grafos o redes residuales
- Caminos de aumento de flujo
- Cortes

El algoritmo de Ford-Fulkerson empieza desde un flujo nulo f_o y en cada iteración se aumenta el flujo utilizando un camino de aumento de la **red residual** que explicaremos luego. Además el algoritmo termina cuando ya no hay caminos de aumento para el flujo calculado. Veamos entonces el algoritmo

Algorithm 12: Ford-Fulkerson

```
Input:
    G = (V, E)
    s, t ∈ V
Flujo f = 0
while Existe un camino de aumento p en la red residual G_f do
    aumentar el flujo f a lo largo de p
return f
```

para entender el algoritmo entonces tenemos que entender primero que es la red residual y luego que es un camino de aumento en esa red.

8.2.1 Capacidades residuales

Para aumentar un flujo f necesitamos cambiar obviamente el flujo existente en algunas aristas del grafo, sin embargo esto no significa que el flujo que pasa por una arista crezca, de hecho puede decrecer. El grafo residual de un gafo $G = (V, E)$ de capacidades c con un flujo f, $G_f = (V, E_f)$, tiene una función de capacidad $c_f : V \times V \to \mathbb{R}^{\geq 0}$

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) \text{ si}(u,v) \in E \\ f(v,u) \text{ si}(v,u) \in E \\ 0 \text{ en otro caso} \end{cases}$$

Veamos entonces que en G_f si es posible tener las aristas (u,v) y (v,u). Admeás por cada arista $(u,v) \in E$ se tienen dos aristas $(u,v),(v,u) \in E_f$ y por consecuencia $|E_f| \leq 2 \cdot |E|$. Además el grafo G_f junto con c_f forman un problema de flujo en redes.
Si f es un flujo en G y f' un flujo en G_f. El aumento de f por f' denotado como $f \uparrow f'$ es

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) \text{ si} (u,v) \in E \\ 0 \text{ en otro caso} \end{cases}$$

Lema 3

Sea $G = (V,E)$ una red de flujo y f un flujo sobre G. Sea G_f la red residual para f y f' un flujo sobre G_f. Entonces, $f \uparrow f'$ es un flujo sobre G con valor $|f| \uparrow |f'|$

: *Para demostrar este lema hay que demostrar entonces 3 cosas:*

- *Verificar que $f \uparrow f'$ satisface las condiciones de capacidad: Si $(u,v) \notin E$ entonces $f \uparrow f'(u,v) = 0$. Sino $(u,v) \in E$, veamos que:*

  $$(f \uparrow f')(u,v) = f(u,v) + f'(u,v) - f'(v,u) \geq f(u,v) + f'(u,v) - f(u,v) = f'(u,v) \geq 0$$

  Además también se observa que:

  $$(f \uparrow f')(u,v) = f(u,v) + f'(u,v) - f'(v,u) \leq f(u,v) + f'(u,v) \leq f(u,v) + c_f(u,v) = c(u,v)$$

- *Conservación de flujo:*

  $$\sum_v (f \uparrow f')(u,v) = \sum_v f(u,v) + f'(u,v) - f'(v,u) = \sum_v f(v,u) + f'(v,u) - f'(u,v) = \sum_v (f \uparrow f')(v,u)$$

- *Calculo del valor de flujo: Definimos dos conjuntos* ***disjuntos*** $V_1 = \{v \in V : (s,v) \in E\}$ *y* $V_2 = \{v \in V : (v,s) \in E\}$*, estos son disjuntos por la inexistencia de aristas paralelas. Luego*

  $$|f \uparrow f'| = \sum_{v \in V_1} (f \uparrow f')(s,v) - \sum_{v \in V_2} (f \uparrow f')(v,s) = (f \uparrow f')(s, V_1) - (f \uparrow f')(V_2, s)$$

  Desglosando y reordenando ambos terminos nos queda

  $$= f(s,V_1) - f(V_2,s) + f'(s,V_1) + f'(s,V_2) - f'(V_1,s) - f'(V_2,s) = |f| + |f'|$$

  ■

Definición 8.2.1: Caimno de aumento

ada una red de flujo $G = (V, E)$ y un flujo f, un camino de aumento es un camino simple p de s a t en la red residual. Ese camino define una función de flujo f_p sobre la red residual que por el lema anterior forma un flujo $|f| + |f_p| = |f| + c(p)$, donde $c(p)$ es la menor capacidad de un arco en p. Dado que p es un camino en G_f esto también significa que $c(p) > 0$

Dado esto, entonces nuestro algoritmo de ford fulkerson quedaría de la siguiente manera

Algorithm 13: Ford-Fulkerson con caminos de aumento

```
Input:
    G = (V, E)
    s, t ∈ V
Flujo f = 0
while Existe un camino de aumento p en la red residual G_f do
    c(p) = min{c_f(u, v) : (u, v) ∈ p}
    for (u, v) ∈ p do
        if (u, v) ∈ E then
            f(u, v) = f(u, v) + c(p)
        else
            f(u, v) = f(u, v) − c(p)
return f
```

Entonces ahora tenemos un algoritmo que va aumentando y el flujo cada vez mientras exista un camino en la red residual, pero tenemos que demostrar que cuando ya no se encuentren caminos, entonces es porque el algoritmo ha entontrado el flujo máximo, para esto necesitamos introducir el concepto de **Cortes en una red de flujo**.

8.2.2 Cortes en una red de flujo

Un corte (también definido en la sección de arboles cobertores) es una partición del conjunto de nodos del grafo en dos conjuntos (S, T), con $S \cup T = V$. En una red de flujo, el corte cumple que $s \in S$ y $t \in T$, es decir que pertenecen a conjuntos distintos. El flujo neto que cruza ese corte es entonces $f(S, T) - f(T, S)$, es decir el flujo neto que cruza por las aristas que conectan nodos de los diferentes conjuntos.

La capacidad de un corte (S, T) es simplemente

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

Y un corte (S, T) es mínimo s ies de minima capacidad entre todos los posible cortes, es decir, sea (S', T') un corte cualquiera, entonces (S, T) es de minima capacidad si $c(S, T) \leq c(S', T')$. Se observa que un flujo que pasa por un corte no puede sobrepasar la capacidad del corte y esto lo formalizamos a continuación.

Lema 4

Sea f un flujo sobre una red $G = (V, E)$ con vertices s y t. Sea (S, T) un corte de G. Entonces, el flujo neto a través del corte es igual a $|f|$, es decir $f(S, T)* = |f|$

:

$$
\begin{aligned}
|f| &= \sum_v f(s,v) - \sum_v f(v,s) + \sum_{u\in S\setminus\{s\}} \left(\sum_v f(u,v) - \sum_v f(v,u)\right) \\
&= \sum_v \big(f(s,v) + f(S\setminus\{s\},v)\big) - \sum_v \big(f(v,s) + f(v,S\setminus\{s\})\big) \\
&= \sum_v f(S,v) - \sum_v f(v,S) \\
&= f(S,S) + f(S,T) - f(S,S) - f(T,S) \\
&= f(S,T) - f(T,S) = f(S,T)^* \blacksquare
\end{aligned} \tag{1}
$$

Esto nos da como consecuencia directa el siguiente corolario

Corolario 3

Dado un flujo f y un corte (S,T) entonces $|f| \le c(S,T)$

Con esto llegmaos al teorema que nos permitirá confirmar que Ford Fulkerson es correcto.

Teorema 8.2.2: *Sea f un flujo sobre una red $G=(V,E)$ con vertices s y t. Las siguientes condiciiones son equivalentes:*

1. *f es un fllujo máximo sobre G*
2. *No existe camino de aumento en la red residual G_f*
3. *$|f| = c(S,T)$ para algún corte (S,T) de G*

: *Demostraremos el teorema demostrando* $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1$*:*

- $1 \Rightarrow 2$*: De existir un camino de aumento en* G_f *esto quería decir que* f *no es un flujo máximo, por lo tanto es una contradicción.*
- $3 \Rightarrow 1$*: Por el corolario anterior, un flujo siempre es menor a la capacidad de un corte cualquiera. En particular si el flujo es igual a la capacidad de un corte* (S,T)*, es decir que* $|f| \le c(S,T)$ *entonces es imposible aumentarlo, de lo contralio romperíamos con la condición del corolario anterior de que un flujo es menor a la capacidad de un corte cualquiera.*
- $2 \Rightarrow 3$*: Definamos como* $S = \{u : \text{existe un camino de } s \text{ a } u \text{ en } G_f\}$*. Consideremos el corte* $(S, V \setminus S)$*, sea* $u \in S$ *y* $v \in T$*. Sabemos que si:*
  - $(u,v) \in E$ *entonces* $f(u,v) = c(u,v)$
  - $(v,u) \in E$ *entonces* $f(v,u) = 0$
  - *sino* $f(u,v) = f(v,u) = 0$

  Con lo cual, nos queda entonces lo siguiente

$$
\begin{aligned}
|f| &= f(S,T)^* = f(S,T) - f(T,S) \\
&= \sum_{u \in S} \sum v \in T f(u,v) - \sum v \in T \sum u \in S f(v,u) \\
&= \sum_{u \in S} \sum v \in T c(u,v) - \sum v \in T \sum u \in S 0 \\
&= c(S,T) \blacksquare
\end{aligned}
\tag{2}
$$

Luego, esto quiere decir que la única forma que no existan caminos de aumento es que el flujo sea máximo y esto implica directamente la correctitud de el algoritmo de Ford Fulkerson.

Corolario 4

El algoritmo de Ford-Fulkerson **termina**, el flujo retornado es de valor máximo

Corolario 5

Ford-Fulkerson **termina** tiene un tiempo de complejidad de $O(|E| \times |f^*|)$, donde f^* es el flujo máximo de un grafo

El corolario anterior es una consecuencia directa de que a lo sumo se encontrarán $|f^*|$ caminos de aumento, y a lo sumo cada camino de aumento tiene $|E|$ aristas.

9

SECCIÓN

Miscellaneous

9.1 Rule

`\rule`{*length*}{*thickness*}

This command inserts a horizontal line with the designated width and thickness.

`\hrulefill`

Inserts a horizontal line that spans the entire width of the page.

`\vrule`{*height*}{*thickness*}

This command inserts a vertical line with the designated width and thickness.

10

SECCIÓN

Some Common Errors

- A command was misspelled and LATEX doesn't recognize it.
- You have a special character

  `(e.g., &, $, #, %) in the text. Enter these as \&, \$, \# or \%.`

- `Whatever you \begin{...}, you must \end{...}. This includes braces: { ... }.`
- A command was not given all its arguments and is trying to use the rest of the file for an argument.

SECCIÓN 11

Further Reading

11.1 Websites and Tutorials

If you are interested in understanding how Latex really works, I highly encourage reading the following package pdfs/websites/tutorials:
If you are looking for a solution for a problem you have encountered and can not find it within these files, see the templates created in the folder that orignially contained this pdf file or contact me at: armindubert19@gmail.com

11.2 Introduction/Commands/and Overall Good References

11.2.1 Writing First Document

- http://ctan.org/pkg/first-latex-doc

  Writing your first document, with a bit of both text and math.

11.2.2 Brief Introduction/List of Commands

- https://www.scss.tcd.ie/~dwoods/1617/CS1LL2/HT/wk1/commands.pdf

  This online article is an extremely brief reference guide for some Latex Commands.

- https://www.bu.edu/math/files/2013/08/LongTeX1.pdf

  List of several useful commands.

11.2.3 In-Depth LaTeX Guide

- https://tobi.oetiker.ch/lshort/lshort.pdf

  This online article is a semi-lengthy but thorough introduction to LaTeXĠreat for people who are just starting.

- https://www2.mps.mpg.de/homes/daly/GTL/gtl_20030512.pdf

  Another extensive guide that does a good job teaching the basics as well as go a little more in depth.

- http://www.rpi.edu/dept/arc/docs/latex/latex-intro.pdf

  Another extensive guide that does a good job teaching the basics as well as going a little more in depth in some areas.

11.3 Boxes

While this source is long, it provides great insight that leads to a greater understanding of the entire LaTeX system. So if you are eager to dig deep into the latex language, the following source is a great place to start.

- tcolorbox.pdf

  This source is contained in the folder along with this file. This source is the longest source out of all the sources but is packed with information about all the great things you can do with boxes, such as the ones seen in this guide.

11.4 Typesetting Math Documents

11.4.1 Creating Math Documents

- http://www-math.mit.edu/~psh/exam/examdoc.pdf

  Extensive guide on how to write exams.

- mathexam.pdf

  You will find this source in the mathexam folder after having downloaded the mathexam.zip file onto your computer. This is a great source if you want to learn an alternate way to creating math test/exam using latex.

11.4.2 Symbols

- https://www.caam.rice.edu/~heinken/latex/symbols.pdf

  List of Mathematical Symbols.

- https://www.bu.edu/math/files/2013/08/LongTeX1.pdf

  An additional list of Symbols.

11.4.3 Typing Math Functions/Equations

- https://www.overleaf.com/learn/latex/Integrals,_sums_and_limits

  Goes over typesetting Integrals, Summations, and Limits.

- https://www.overleaf.com/learn/latex/Subscripts_and_superscripts

  Goes over typesetting subscripts and superscripts

- https://www.overleaf.com/learn/latex/Fractions_and_Binomials

  Goes over typesetting fractions and binomials.

11.5 Graphics

11.5.1 Images

- graphicx.pdf

  This source provides a great introduction for people interested in exploring the intricacies behind inserting images into a document.

- https://www.overleaf.com/learn/latex/Inserting_Images

  Goes over some of the specifics behind setting up the `\graphicspath` and inserts Images.

11.5.2 Colors

- http://latexcolor.com/

  Provides you with a multiple of RGB color definitions for the user to use in their documents

11

SECCIÓN

References

[1] Academic and Research Computing. *Text Formatting with LaTeX A Tutorial*. NY, April 2007.
http://www.rpi.edu/dept/arc/docs/latex/latex-intro.pdf

[2] Leslie Lamport. *LaTeX for Beginners*. fifth edition, Document Reference: 3722-2014, March 2014.
http://www.docs.is.ed.ac.uk/skills/documents/3722/3722-2014.pdf

[3] Tobias Oetiker, Hubert Partl, Irene Hyna, and Elisabeth Schlegl. *The Not So Short Introduction to LaTeX*. Version 6.3, March 1994.
https://tobi.oetiker.ch/lshort/lshort.pdf

[4] Helmut Kopka and Patrick W. Daly. *A Guide to LaTeX and Electronic Publishing*. Addison-Wesley, fourth edition, May 2003.
https://www2.mps.mpg.de/homes/daly/GTL/gtl_20030512.pdf

[5] Philip Hirschhorn. *Using the exam document class*. Wellesley College, second edition, MA, November 2017.
http://www-math.mit.edu/~psh/exam/examdoc.pdf

[6] *When should I use \input vs \include*.
https://tex.stackexchange.com/questions/246/when-should-i-use-input-vs-include

[7] Overleaf. *Inserting Images*.
https://www.overleaf.com/learn/latex/Inserting_Images

[8] Rice University. *LaTeX Mathematical Symbols*.
https://www.caam.rice.edu/~heinken/latex/symbols.pdf

[9] Overleaf. *Integrals, sum and limits.*
https://www.overleaf.com/learn/latex/Integrals,_sums_and_limits

[10] Boston University. *LaTeX Command Summary*. December 1994.
https://www.bu.edu/math/files/2013/08/LongTeX1.pdf

[11] Overleaf. *Subscripts and superscripts.*
https://www.overleaf.com/learn/latex/Subscripts_and_superscripts

[12] Disquis. *LaTeX Color*. Addison-Wesley, second edition, Reading, MA, 1994.
http://latexcolor.com/

[13] David Woods. *Useful LaTeX Commands.*
https://www.scss.tcd.ie/~dwoods/1617/CS1LL2/HT/wk1/commands.pdf

[14] Overleaf. *Spacing in Math Mode.*
https://www.overleaf.com/learn/latex/Spacing_in_math_mode

[15] Overleaf. *Fractions and Binomials.*
https://www.overleaf.com/learn/latex/Fractions_and_Binomials

[16] Overleaf. *Environments.*
https://www.overleaf.com/learn/latex/Environments

[17] Overleaf. *Margin Notes.*
https://www.overleaf.com/learn/latex/Margin_notes

[18] Art of Problem Solving. *LaTeX:Commands*
https://artofproblemsolving.com/wiki/index.php/LaTeX:Commands

[19] Latex-Project.
https://www.latex-project.org/about/

[20] Tex Stack Exchange. customizing part style with Tikz.
https://tex.stackexchange.com/questions/159551/customizing-part-style-with-tikz

[21] Tex Stack Exchange. How to change chapter/section style in tufte-book?.
https://tex.stackexchange.com/questions/83057/how-to-change-chapter-section-style-in-
noredirect=1&lq=1