Algoritmos III - Apuntes para final

Gianfranco Zamboni

12 de julio de 2022

Índice

1. Introducción

Un **algoritmo** es una secuencia **finita** de pasos **precisos** (no deben requerir tomar ninguna decisión subjetiva, ni hacer uso de la intuición o la creatividad) necesarias para llevar a cabo un cálculo de manera correcta.

Se dice que un algoritmo está **bien definido** cuando toda ejecución del mismo bajo los mismos parámetros devuelve el mismo resultado. Hay que tener en cuenta que si bien esto es lo deseable en la mayoría de los casos, muchas veces es necesario usar variables aleatorias o heurísiticas para conseguir un resultado medianamente decente para problemas que no tienen solución o que son muy dificiles de calcular de manera exacta.

Pseudocódigo: Es una descripción informal de alto nivel de un algoritmo que transmite el procedimiento a realizar de forma clara y precisa.

1.1. Analisis de algoritmos

Cuando tenemos más de un algoritmo para resolver un problema y queremos elegir el *mejor* entre ellos, hay diferentes criterios que podemos analizar para medir la eficiencia de los mismos, según el contexto de aplicación. Los recursos de mayor interés suelen ser el tiempo de cómputo y el espacio de almacenamiento requerido. Podemos considerar dos enfoques:

Análisis empírico: Implementar el algoritmo en una máquina determinada utilizando un lenguaje determinado, correrlo para un conjunto de instancias y comparar sus tiempos de ejecución. Esto implicaría perder tiempo y esfuerzo programandolo, ejecutandolo y además probarlo con un conjunto de instancias acotados por lo que realmente no podríamos concluir nada sobre su comportomiento con instancias que queden fuera de ese conjunto.

Análisis teórico: Determinar matemáticamente la cantidad de tiempo que llevará su ejecución como una función de la medidad de la instancia considerada, independinzándolo de la máquina sobre la cuál es implementado el algoritmo y el lenguaje para hacerlo. Para esto es necesario definir un modelo de cómputo, un lenguaje sobre este modelo, cuales son las instancias del problema relevantes al mismo y sus tamaños.

Dicho de otra forma, desde el punto de vista teórico, se busca determinar la velocidad de crecimiento del tiempo o el espacio requerido por el algoritmo en función del tamaño de las instancias del problema.

Dado un problema, se le asigna un entero n (llamado el **tamaño** del problema) que servirá como medida de la cantidad de datos de entrada. Y se define como **complejidad temporal** del algoritmo al tiempo ejecución de un algoritmo expresado como una función que depende de n.

Análogamente, se puede definir la **complejidad espacial** del algoritmo como el espacio de memoria necesario para su ejecución en función del tamaño del problema.

1.1.1. Modelo Random Access Machine (RAM)

Una Random Access Machine (RAM) modela una computadora con un único registro acumulador en la cuál las instrucciones no pueden modificarse. La misma está formada por:

- Una **unidad de entrada** que contiene los datos necesarios para poder ejecutar una corrida del algoritmo. La misma es de solo lectura y, en cada una de sus celdas, puede contener un entero de tamaño arbitrario. Cuando un valor es leído, la cinta se avanza una celda.
- Una **unidad de salida** que es de solo escritrura y se encuentra inicialmente vacía. Cuando una instrucción es ejecutada, se imprime un entero en la celda que se encuentra bajo la lectora y luego se la avanza una posición.
- Una **memoria** que consiste en una secuencia infinita de registros, cada uno de los cuales puede contener un entero de tamaño arbitrario.
- Un **programa** es una secuencia de instrucciones etiquetadas que no está almacenado en memoria. Asumimos que las instrucciones que contiene un programa, son las instrucciones aritméticas básicas, de entradas/salidas y direccionamiento indirecto y de branching encontradas en la mayoría de las computadoras.

Este modelo se usa cuando los tamaños de los problemas a resolver es lo suficientemente pequeño como para que entren en la memoría principal de una computadora o cuando los enteros usados en la computación son los suficientemente chicos como para que entren en una palabra de la misma.

1.2. Cálculo de complejidad

Una operación es elemental si su tiempo de ejecución puede ser acotado por una constante dependiente sólo de la implementación particular utilizada. Esta constantes no depende de la medidad de los parámetros de la instancia considerada.

En una máquina RAM, se asume que toda instrucción es una operación elemental con un **tiempo de ejecución** asociado y definimos el tiempo de ejecución de un programa como:

$$t_A(I) = \text{suma de los tiempos de ejecución de las instrucciones realizadas por el programa } A \text{ con la instancia } I$$

1.2.1. Tamaño de una instancia

Para especificar la complejidad en peor caso (tanto espacial como temporal), debemos especificar el tiempo de ejecución requerido por cada instrucción y el espacio de cada registro.

La forma más simple para esto es usar el **criterio de costo uniforme** en el que cada instrucción requiere una unidad de tiempo y cada registro requiere una unidad de espacio. En general, usaremos este criterio para analizar problemas de ordenamientos o sobre grafos y matrices.

Una segunda opción (un poco más realista), es tomar en cuenta el tamaño real (en bits) de los operandos. Este el **criterio de costo logarítmico** que toma en cuenta que son necesarios para representar los datos.

Alfabeto: Es el conjunto de símbolos que puede interpretar en una máquina. Por ejemplo, las computadoras actuales reconocen el alfabeto binario $\{0, 1\}$.

Dada una instancia I, se define $|I|$ como el número de símbolos de un alfabeto finito necesarios para codificar I. Este tamaño depende del alfabeto elegido. En el caso de las computadoras actuales, que se manejan con el alfabeto binario, para almacenar un número natural n, se necesitan $L(n) = \lfloor \log_2 n \rfloor + 1$ bits.

Usaremos este criterio cuando analicemos problemas sobre números (por ejemplo, el calculo de un factorial).

1.2.2. Funciones de complejidad

Complejidad peor caso: Se toma como complejidad del algoritmo, la función que dada una instancia de tamaño de n, devuelve la máxima complejidad posible sobre todas las instancias de ese tamaño.

Complejidad caso promedio: Se toma como complejidad del algoritmo la función que dada una instancia de tamaño de n, devuelve el promedio de todas las complejidades posible sobre todas las instancias de ese tamaño.

Dadas dos funciones $f, g : \mathbb{N} \to \mathbb{R}$ decimos que:

- $f(n) = O(g(n))$ si existen $c \in \mathbb{R}_+ y n_0 \in \mathbb{N}$ tales que

$$f(n) \leq cg(n) \text{ para todo } n \geq n_0$$

- $f(n) = \Omega(g(n))$ si existen $c \in \mathbb{R}_+ y n_0 \in \mathbb{N}$ tales que

$$f(n) \geq cg(n) \text{ para todo } n \geq n_0$$

- $f(n) = \Theta(g(n))$ si

$$f(n) = O(g(n)) \text{ y } f(n) = \Omega(g(n))$$

1.2.3. Problemas bien resueltos

Decimos que un problema está **bien resuelto** si existe un algoritmo de complejidad polinomial que lo resuelva. Es decir, que no consideraremos soluciones sastifactorios a los algoritmos supra-polinomiales.

1.3. Técnicas de diseño de algoritmos

1.3.1. Algoritmos golosos

Los algoritmos golosos toman decisiones basandose en la información disponible actualemente, sin considerar los efectos que esas decisiones podrían tener en el fúturo. Son fáciles de inventar, implementar y son eficientes. Sin embargo, aunque muchas veces proporcionan heurísticas sencillas para resolver problemas de optimización, no siempre funcionan.

Comunmente, los algoritmos golosos y los problemas que pueden resolver tienen las siguientes carácteristicas:

- El problema se debe resolver de alguna manera óptima. Para construir su solución se mantiene **un conjunto de candidatos** entre los cuales el algoritmo puede elegir para agregar a la solución en el próximo paso. Por ejemplo, un conjunto de monedas o de nodos de un grafos.
- A medida que el algoritmo progresa, se acumulan dos conjuntos: Uno contiene candidatos que ya fueron considerados y usados; el otro contiene los candadidatos que fueron consiradosy rechazados.
- Hay una **función de verificación** que dado un conjunto de candidatos verifica si puede ser una solución valida al problema propuesto auque esta no sea la solución óptima.
- Hay una **función de factibilidad** que dado un conjunto de candidatos verifica si se puede extender con nuevos candidatos para conseguir una solución al problema.
- Hay una **función de selección** que decide cual de los candidatos disponibles (que no fueron elegidos ni rechazados) es el más prometedor para hacercanos a la solución deseada.
- Hay una f**función objetivo** que nos da el valor de la solución encontrada. Por ejemplo, el número de monedas utilizadas para el cambio, o la cantidad de nodos usados en un cámino, o cualquier otro valor que estemos intentando optimizar. A diferencia de las otras tres funciones mencionadas anteriormente, esta no aparece explicitamente en el algoritmo goloso.

Estructura general de un algoritmo goloso:

```
function GREEDY(C: Set)                          ▷ C es el conjunto de candidatos
    S ← ∅                                        ▷ Conjunto en el que se construye la solución
    while C ≠ ∅ ∧ ¬esSolucion(S) do
        x ← SELECCIONAR(C)
        C ← C \ {x}
        if esValido(S ∪ {x}) then
            S ← S ∪ {x}
        end if
    end while
```

```
    if esSolucion(S) then return S
    else
        return Error: No hay solución
    end if
end function
```

1.3.2. Recursividad

Un algoritmo recursivo se define en términos de si mismo, esto es, en su cuerpo aparece una aplicación suya. En general, las llamadas recursivas se aplican sobre párametros más pequeños que los iniciales. Cuando el parámetro sobre el que se aplica es lo suficientemente chico se ejecuta lo que se llama el **caso base** que, para su resolución, no necesita llamar a la misma función, terminando así el proceso recursivo.

En general, para calcular la complejidad de un algoritmo recursivo, se debe plantear primero las ecuaciones de recurrencia y luego, utilizando herramientas matemáticas, encontrar la fórmula cerrada (que no dependa de la complejidad de instancias más chicas) de esta ecuación.

1.3.3. Divide and Conquer

Es una técnica que consiste en dividir la instancia de un problema a resolver en varias instancias más pequeñas del mismo problema y resolverlas de manera independiente para luego combinar sus soluciones en la solución de la instancia original.

Para que valga la pena realizar un algoritmo con esta técnica se deben cumplir tres condiciones:

1. Debe ser posible descomponer las instancias en subinstancias y combinar las subsoluciones de manera eficiente.

2. Las subinstancias deben ser todas más o menos del mismo tamaño. La mayoría de los algoritmos de divide and conquer son tales que el tamaño de una subinstancia I, es aproximadamente $\frac{n}{b}$ donde n es el tamaño de la instancia original y b alguna constante.

La estructura general de un algortimo de divide and conquer es la siguiente:

```
function DivideAndConquer(I: Instancia del problema)
    if I es suficientemente pequeño o simple then return resolver(I)
    else
        Descomponer I en subinstancias I_1, I_2, ..., I_k
        for i ← 1 to k do
            y_i ← DivideAndConquer(I_i)
        end for
        Combinar las soluciones y_1, y_2, ..., y_k obtenidas para obtener la solución y de I
        return y
    end if
```

end function

En este tipo de algoritmos, generalmente las instancias que son caso base toman tiempo constante c y para las casos recursivos se pueden identificar tres puntos críticos:

- Cantidad r de llamadas recursivas que haremos.
- Medida de cada subproblema: $\frac{n}{b}$ para alguna constante b.
- Tiempo requerido por el algoritmo para ejecutar, descomponer y combinar para una instancia de tamaño n, $g(n)$.

Entonces, el tiempo total $T(n)$ consumido por el algoritmo está definido por la siguiente ecuacución de recurrencia:

$$T(n) = \begin{cases} c & \text{si } n \text{ es caso base} \\ rT\left(\frac{n}{b}\right) + g(n) & \text{si } n \text{ es caso recursivo} \end{cases}$$

1.3.4. Backtracking

Es una técnica que permite recorrer sistemáticamente todas las posibles configuraciones del espacio de soluciones de un problema computacional. Cuando el algoritmo comienza, no se sabe nada sobre la solución del problema. A medida que va avanzando, se agrega un elemento a la solución para ir consiguiendo soluciones parciales. Si en algún momento, la solución parcial deja de poder ser extendida entonces esa solución se descarta.

La idea básica es tratar de extender una solución parcial del problema hasta, eventualemnte, llegar a obtener una solución completa, que podría ser válida o no. Habitualmente, se utiliza un vector $a = (a_1, a_2, \ldots, a_n)$ para representar una solución candidata donde cada a_i pertenece a un dominio finito A_i. El espacio de soluciones es el producto cartesiano $A_1 \times \cdots \times A_n$.

En síntesis, en cada paso se extienden las soluciones parciales $a = (a_1, a_2, \ldots, a_k)$ con $k < n$, agregando un elemento más al final. Las nuevas soluciones parciales son sucesoras de la anterior.

Se puede pensar este espacio de búsqueda como un árbol dirigido donde cada vértice representa una solución parcial y un vértice x es hijo de y si la solución parcial x se puede extender desde la solución parcial y. La raíz del árbol se corresponde con el vector vacio.

El proceso de bactkracking recorre este árbol en profundidad. Cuando se puede deducir que una solución parcial no llevará a una solución válida, no es necesario seguir explorando esa rama del árbol y se retrocede hasta encontrar un vértice con un hijo válido por donde seguir la exploración.

El template general para algoritmos de este tipo es:

```
sol: Vector ← []
encontro: Booleano ← false
function BACKTRACKING(v[1...k], s: Instancia del problema)
    if k == 0 ∧ esSolucion(s) then
        sol ← s
```

```
        encontro ← true
    else
        for i ← 1 to k do
            if encontro then return
            end if
        end for
    end if
end function
```

1.3.5. Programación dinámica

Esta técnica es aplicada a problemas de optimizacióncombinatoria, donde puede haber muchas soluciones fáctibles, cada una con un valor (o costo asociados) y pretende obtener la solución con mejor valor (o menor costo).

Al igual que dividir y conquistar, se divide al problema en problemas que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original. Se diferencian, en que esta técnica es iterativa (no recursiva) y es adecuada cuando es necesario resolver varias subisntancias del problema que son iguales ya que se van almacenando los resultados parciales obtenidos para su posterior utilización.

Principio de Optimalidad de Bellman: Un problema de optimización satisface este principio si en una solución óptima cada subsolución es a su vez óptima del subproblema correspondiente. Es decir, dada una secuencia óptima de decisiones, todas subsecuencia de ella es, a su vez, óptima.

Este principio es condición necesaria del problema para poder usar la técnica.

1.3.6. Algoritmos probabilísticos

Cuando un algoritmo tiene que hacer una elección, a veces es preferible elegir al azar en vez e gastar mucho tiempo tratando de ver cual es la mejor opción.

Una de las características de los algoritmos probabilisticos es que puede comportarse de maneras distintas si se lo usa para computar una misma instancia dos o más veces. Su tiempo de ejecución y resultados pueden variar considerablemente de un uso a otro pudiendo incluso nunca llegar a terminar en ciertas corridas para una instancia específica. Sin embargo, dada la posibilidad de reiniciar el algoritmo y obtener un resultado válido éste es un comportamiento que no molesta.

Por otro lado, una consecuencia de este comportamiento es que, si hay más de una solución al problema, las mismas se pueden obtener corriendo el algoritmo más de una vez.s

Los algoritmos probabílisticos se pueden clasificar dependiendo de la probabilidad de que devuelvan una respuesta correcta:

- **Algoritmos númericos**: Estos algoritmos devuelven un intervalo de confianza (del estilo "La solución es $x \pm y$ con probabilidad z)". Por lo general, mientras más tiempo de proceso

les demos, mas preciso es el intervalo que devuelven.

- **Algoritmos de Montecarlo**: Son algoritmo que dan la respuesta exacta con alta probabilidad. Por lo general, no es posible verificar que la respuesta dada sea correcta sin embargo la probabilidad de error disminuye mientras más tiempo se este ejecutando el algoritmo.
- **Algoritmos Las Vegas:** Son algoritmos que siempre dan una respuesta correcta pero puede llegar a no dar ninguna respuesta.
- **Algoritmos Sherwood:** Son algoritmos que agregan una componente aleatoria a algoritmos determinísticos para tratar de evitar tiempos de ejecución del peor caso. Un ejemplo de esto es el Quicksort con pivote seleccionado aleatoriamente.

1.3.7. Heurísticas

Dado un problema de optimización $\sqcap$ díficl de resolver (y para el cual probablemente no exista un algoritmo eficiente), los **algoritmos heurísticos** definen un procedimiento que intenta conseguir soluciones para el mismo pero puede devolver resultados erróneos o no devolver nada directamente.

En otras palabras: Sea I es una instancaia del problema y $x^*(I)$ el valor óptimo de la función a optmizar en dicha instancia. Buscamos crear una heurística H tal que la solución $x^H(I)$ devuelta por la misma sea lo más cercano a $x^*(I)$ posible.

Algoritmos apróximados: Decimos que H es un algoritmo $\epsilon-$aproximado para el problema $\sqcap$ si para algún $\epsilon > 0$ vale que $|x^H(I) - x^*(I) \leq \epsilon|x^*(I)|$

Algoritmos con certificados: Por lo general, si bien los problemas dificiles no pueden ser resueltos con algoritmos polinomiales si se puede verificar que las soluciones provistas por las heurísticas sean correctas con algoritmos polinomiales.

Se dice que los algoritmos diseñados para realizar esta verificación proveen un certificado que afirma la validez de la soluciones propuestas por una heurística.

2. Grafos

2.1. Definiciones básicas

Los grafos proporcionan una forma conveniente y flexible de representar problemas de la vida real que consideran una red como estructura subyacente. Esta red puede ser física (como instalaciones eléctticas) o abstractas (que modelan relaciones menos tangibles, como relaciones sociales y bases de datos).

Matemáticamente, un grafo $G = (V, X)$ es un par de conjuntos, donde V es un conjunto de **puntos / nodos / vértices** y X es un subconjunot del conjunto de pares no ordenados de elementos distintos de V. Los elementos de X se llamas **aristas, ejes o arcos**.

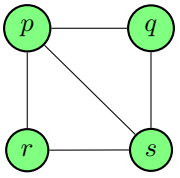


Figura 1: $G = ([p,\ q,\ r,\ s],\ [(p,q),\ (p,s),\ (q,s),\ (r,s)])$

Dados v y $w \in V$, si $e = (v, w) \in X$ se dice que v y w son **adyacentes** y que e es **incidente** a v y a w.

La definición de grafo no alcanza para modelar todas las situaciones posibles de una red. Por ejemplo, si se quisiese modelar la ruta de los aviones entre varias ciudades, deberiamos poder modelar varios vuelos entre dos ciudades:

Multigrafo: Es un grafo en el que puede haber varias aristas entre el mismo par de nodos distintos.

Figura 2: Multigrafo

Seudografo: Es un grafo en el que puede haber varias aristas entre cada par de nodos y también puede haber aristas (*loops*) que unan a un nodo con sí mismo.

Figura 3: Multigrafo

Notación: $n = |V|$ y $m = |X|$

Grado: El grado $d_G(v)$ de un nodo v es la cantidad de aristas incidentes a v en el grafo G. Notaremos $\Delta(G)$ al máximo grado de los vértices de G y $\delta(G)$ al mínimo.

Nota: En un seudografo, un loop aporta 2 al grado del vértice.

Teorema 1. *La suma de los grados de los nodos de un grafo es igual a dos veces el número de aristas, es decir:*

$$\sum_{i=1}^{n} v_i = 2m$$

DEMOSTRACIÓN

Sea $G = (V, X)$ un grafo de n nodos y m aristas. Haremos inducción en la cantidad de aristas:

Caso base: $m = 1$.
En este caso, el grafo G tiene sólo una arista que notamos $e = (u, w)$. Entonces $d(u) = d(w) = 1$ y $d(v) = 0$ para todo $v \in V$ tal que $v \neq u, w$. Por lo tanto,

$$\sum_{v \in V} d(v) = 2$$

y $2m = 2$, cumpliendose la propiedad.

Paso inductivo: Consideremos un grafo $G = (V, X)$ con m aristas ($m > 1$). Nuestra hipotesis inductiva es: *En todo grafo $G' = (V', X')$ con m' aristas ($m' < m$), se cumple que*

$$\sum_{v \in V'} d_{G'}(v) = 2m'$$

Sea $e = (u, w) \in X$, llamemos $G'' = (V, X - \{e\})$ al grafo que resulta si le quitamos e a G. Como la cantidad de aristas de G'' es $m - 1$, G'' cumple con la hipótesis inductiva. Entonces vale que

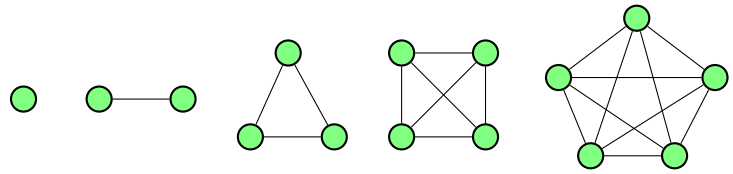
$$\sum_{v \in V} d_{G''}(v) = 2(m - 1)$$

Como $d_G(u) = d''_G(u) + 1$, $d_G(w) = d''_G(w) + 1$ y $d_G(x) = d''_G(x)$ $\forall\ x \in V,\ x \neq u, w$, obtenemos que:

$$\sum_{v \in V} d_G(v) = \sum_{v \in V} d_{G''}(v) + 2 = 2(m - 1) + 2 = 2m$$

Corolario 1. *La cantidad de vértices de grado impar de un grafo es par.*

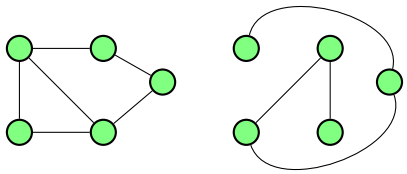
Grafo completo: Son grafos en los que todos sus vértices son adyacentes entre sí. Notaremos como K_n al grafo completo de n vértices.



K_1 K_2 K_3 K_4 K_5

Propiedad: Un grafo completo de n nodos tiene $\frac{n(n-1)}{2}$ aristas

Grafo complemento: Dado un grafo $G = (V, X))$, su grafo complemento $\bar{G} = (V, \bar{X})$ es el grafo con el mismo conjunto de vértices pero un par de vértices son adyacentes en $\bar{G}$ si y solo si no son adyacentes en G.



Grafo G Grafo $\bar{G}$

Propiedad: Si G tiene n vértices y m aristas, entonces:

$$m_{\bar{G}} = \frac{n(n-1)}{2} - m$$

2.1.1. Caminos y ciclos

Camino: Un camino en un grafo, es una secuencia alternada de vértices y aristas

$$P = v_0 e_1 v_1 e_2 \dots v_{k-1} e_k v_k$$

tal que un extremo de la arista e_i es v_{i-1} y el otro es v_i para $i = 1 \dots k$.

Camino simple: Es un camino que no pasa dos veces por el mismo vértice.

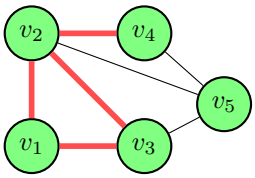
Sección: La sección de un camino $P = v_0 e_1 v_1 e_2 \dots v_{k-1} e_k v_k$ es una subsecuencia

$$v_i e_{i+1} v_{i+1} e_{i+2} \dots v_{j-1} e_j v_j$$

de términos consecutivos de P, y lo notamos como $P_{v_i v_j}$.

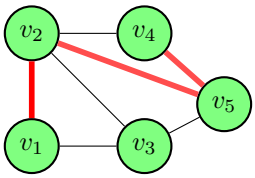
Circuito: Es un camino que empieza y termina en el mismo vértice.

Circuito Simple: Es un circuito de tres o más vértices que no pasa dos veces por el mismo vértices.



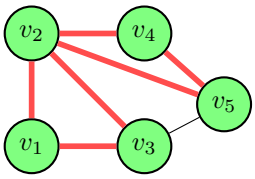
Camino no simple

$P = v_2v_3v_1v_2v_4$



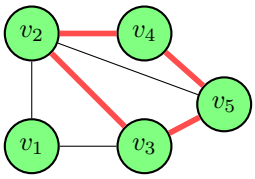
Camino simple

$P = v_1v_2v_5v_4$



Circuito no simple

$P = v_1v_3v_2v_4v_5v_2v_1$



Circuito simple

$P = v_2v_3v_5v_4v_2$

Longitud de un camino: Dado un camino P, su longitud $l(P)$ es la cantidad de aristas que tiene.

Distancia: La distancia $d(v, w)$ entre dos vértices v y w se define como la longitud del camino más corto entre v y w.

- Si no existe camino entre v y w decimos que $d(v, w) = \infty$.
- $\forall\ v \in V,\ d(v, v) = 0$

Proposición 1. *Si un camino P entre v y w tiene longitud $d(v, w)$ entonces P es un camino simple.*

DEMOSTRACIÓN

Demostración por el absurdo. Sea $P = v \dots w$ un camino entre v y w con $l(P) = d(v, w)$. Supongamos que P no es simple, es decir existe un vértice u que se repite en P (u podría llegar a ser v o w) entonces $P = v \dots u \dots u \dots w$.
Formemos ahora un camino $P' = P_{vu}P_{uw}$, como P' no tiene todos los nodos que están en P_{uu} nos queda que $l(P') < l(P) = d(v, w)$. Esto genera un absurdo porque por definición $d(v, w)$ es la longitud del camino más corto entre v y w.

Proposición 2. *La función de distancia cumple las siguientes propiedades para todo u, v, w pertenecientes a V:*

1. $d(u, v) \geq 0$
2. $d(u, v) = 0 \iff u = v$
3. $d(u, v) = d(v, u)$
4. $d(u, w) \leq d(u, v) + d(v, w)$

2.1.2. Subgrafos y Componentes Conexas

Subgrafo: Dado un grafo $G = (V_G, X_G)$, un subgrafo de G es un grafo $H = (V_H, X_H)$ tal que $V_H \subseteq V_G$ y $X_H \subseteq X_G \cap (V_H \times V_H)$. Y notamos $H \subseteq G$.

- Si $H \subseteq G$ y $H \neq G$, entonces H es un **subgrafo propio** de G y notamos $H \subset G$.
- H es un subgrafo generador de G si $H \subseteq G$ y $V_G = V_H$.
- Un subgrafo $H = (V_H, X_H)$ de $G = (V_G, X_G)$, es un **subgrafo inducido** si $\forall\ u, v \in V_H$ tal que $(u, v) \in X_G \Rightarrow (u, v) \in X_H$.
- Un subgrafo inducido de $G = (V_G, X_G)$ por un conjunto de vértices $V' \subseteq V_G$, se denota como $G_{[V']}$.

Grafo G | Sugrafo propio de G | Grafo que no es subgrafo de G | Subgrafo inducido de G

Grafo conexo: Un grafo se dice **conexo** si existe camino entre todo par de vértices.

Componente conexa: Una componente conexa de un grafo $G = (V_G, X_G)$ es un subgrafo conexo maximal de G. Esto es un subgrafo $H = (V_H, X_H)$ inducido de G tal que H es conexo y si tratamos de agregar cualquier vértice $v \in V_G \backslash V_H$ entonces nos queda un grafo no conexo.

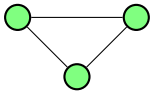
Grafo conexo. | Grafo no conexo. Tiene dos componentes conexas (la verde y la roja)

2.2. Grafos bipartitos

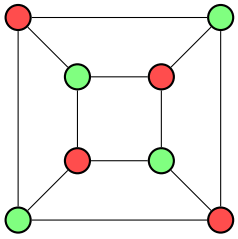
Un grafo $G = (V, X)$ se dice **bipartito** si existe una partición V_1, V_2 del conjunto de vértices V tal que:

- $V = V_1 \cup V_2$
- $V_1 \cap V_2 \neq \emptyset$
- $V_1 \neq \emptyset$
- $V_2 \neq \emptyset$
- Todas las aristas de G tiene un extremo en V_1 y otro en V_2

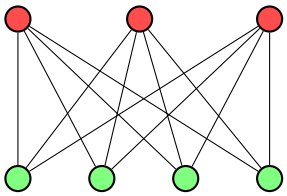
Un grafo bipartito con partición V_1, V_2 es **bipartito completo** si todo vértice en V_1 es adyacente a todo vértice en V_2.



Grafo no bipartito



Grafo bipartito



Grafo bipartito completo

Teorema 2. *Un grafo G con dos o más vértices es bipartito si y solo si no tiene circuitos de longitud impar.*

DEMOSTRACIÓN

Como un grafo es bipartito si y solo si cada una de sus componentes conexas es bipartita alcanza con demostrar el teorema para grafos conexos.

$\Rightarrow$) Sea G un grafo conexo bipartito y $V = (V_1, V_2)$ su bipartición.
Si G no tiene circuitos entonces el teorema se cumple de manera trivial. Supongamos que G tiene circuitos y sea $C = v_1 v_2 \dots v_k v_1$ un circuito de G. Sin perdida de generalidad, supongamos que $v_1 \in V_1$. Como $(v_1, v_2) \in X$ entonces $v_2 \in V_2$). En general, $v_{2i+1} \in V_1$ y $v_{2i} \in V_2$. Como $v_1 \in V_1$ y $(v_k, v_1) \in X$, debe pasar $v_k \in V_2$. Luego $k = 2i$ para algún i, lo que implica que $l(C)$ es par.

⇐) Sea G un grafo conexo sin circuitos impares. Sea u cualquier vértice de V. Definimos:

$$V_1 = \{v \in V \ / \ d(u,v) \text{ es par}\}$$

$$V_2 = \{v \in V \ / \ d(u,v) \text{ es impar}\}$$

V_1 y V_2 definen una partición de V (ya que como G es conexo no hay vértices a distancia ∞ de v). Tenemos que ver que definen una bipartición, es decir que no existe arista entre dos vértice de V_1 y dos de V_2. Hagamos esto por el absurdo:
Supongamos que no es bipartición, es decir existen $v, w \in V_1$ tales que $(v,w) \in X$. Si $v = u$, entonces $d(v,w) = 1$, pero esto no puede pasar por que $d(u,w)$ es par. Lo mismo para w, luego $v \neq u, w$.
Sea P un camino mínimo entre v y u y Q un camino mínimo entre v y w. Como $u, w \in V_1$, P y Q tienen longitud par.
Sea $z \in V$ el último nodo en el que P y Q se cruzan (podría pasar que $z = u$). Como P y Q definen las distancias a v y w respectivamente desde u, entonces P_{uz} y Q_{uz} tienen que ser caminos mínimos. Osea que

$$l(P_{uz}) = l(Q_{uz}) = d(u,z)$$

. Entonces $l(P_{zv})$ y $l(Q_{zw})$ tienen igual paridad. Definamos

$$C = P_{zv}(v,w)Q_{wz}$$

Entonces $l(C) = l(P_{zv}) + l(Q_{wz}) + 1$ que es una longitud impar. Absurdo, partiamos de la supocisión de que G no tenia circuitos de longitud impar.

2.3. Representación de grafos

2.3.1. Matriz de adyacencia de un grafo

Dado un grafo G, se define su **matriz de adyacencia** $A \in \mathbb{R}^{n \times n}$, $A = [a_{ij}]$ como:

$$a_{ij} = \begin{cases} 1 & \text{si } G \text{ tiene una arista entre } v_i \text{ y } v_j \\ 0 & \text{si no} \end{cases}$$

Proposición 3. *Si A es la matriz de adyacencia del grafo G, entonces:*

- *La suma de los elementos de la columna (o fila) i de A es igual a $d(v_i)$.*
- *Los elementos de la diagonal de A^2 indican los grados de los vértices: $a^2_{ii} = d(v_i)$.*

Para los seudografos, se generaliza la definición dada de la seguiente manera:

$$a_{ij} = \begin{cases} \text{cantidad de aristas } (v_i, v_j) & \text{si } i \neq j \\ \text{cantidad de loops sobre } v_i & \text{si } i = j \end{cases}$$

2.3.2. Matriz de incidencia de un grafo

Dado un grafo G, se define su **matriz de incidencia** $B \in \mathbb{R}^{m \times n}$ con $B = [b_{ij}]$ como:

$$b_{ij} = \begin{cases} 1 & \text{si la arista } i \text{ es incidente al vértice } v_j \\ 0 & \text{sino} \end{cases}$$

Proposición 4. *Si B es la matriz de incidencia del grafo G, entonces:*

- *La suma de los elementos de cada fila es igual a 2.*
- *La suma de los elementos de la j-ésima columna es igual a $d(v_j)$.*

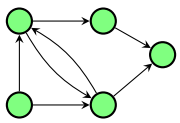
Para los pseudografos, se generaliza la definición de la siguiente forma:

$$b_{ij} = \begin{cases} 2 & \text{si la arista } i \text{ es loop sobre el vertice } v_j \\ 1 & \text{si la arista } i \text{ no es loop e incide sobre el vértice } v_j \\ 0 & \text{sino} \end{cases}$$

2.4. Digrafos

Grafo dirigido o digrafo: Es un par de conjuntos $G = (V, X)$ donde V es el conjunto de nodos y X es un subconjunto del conjunto de pares **ordenados** de elementos distintos de V. A los elementos de X los llamaremos **arcos**.

Dado un arco $e = (u, w)$ llamaremos al primer elemento (u) **cola** de e y al segundo (w), **cabeza** de e.

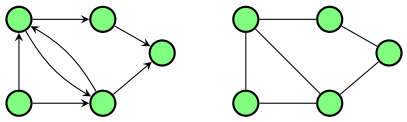


Digrafo G

Grado de entrada: $d_{in}(v)$ de un vértice v de un digrafo es la cantidad de arcos que llegan a v. Es decir, la cantidad de arcos que tienen como cabeza a v.

Grado de entrada: $d_{out}(v)$ de un nodo v de un digrafo es la cantidad de arcos que salen de v. Es decir, la cantidad de arcos que tiene a v como cola.

Grafo subyacente: El grafo subyacente de un digrafo G es el grafo G^S que resulta de remover las direcciones de sus arcos (si para un par de vértices hay arcos en ambas direcciones, sólo se coloca una arista entre ellos).



Digrafo G Grafo G^S

Matriz de adyacencia: de un digrafo G, $A \in \mathbb{R}^{n \times n}$, $A = [a_{ij}]$ se define como:

$$a_i j = \begin{cases} 1 & \text{si } G \text{ tiene un arco } v_i a v_j \\ 0 & \text{si no} \end{cases}$$

Proposición 5. *Si A es la matriz de adyacencia del digrafo G, entonces:*

- *La suma de los elementos de la fila i de A es igual a $d_{out}(v_i)$.*
- *La suma de los elementos de la columna i de A es igual a $d_{in}(v_i)$.*

Matriz de incidencia: de un digrafo G, $B \in \mathbb{R}^{m \times n}$, $B = [b_{ij}]$ se define como:

$$b_i j = \begin{cases} 1 & \text{si } v_j \text{ es cabeza del arco } i \\ -1 & \text{si } v_j \text{ es cola del arco } i \\ 0 & \text{si no} \end{cases}$$

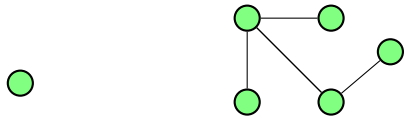
Proposición 6. *Si B es la matriz de incidencia del digrafo G, entonces la suma de los elementos de cada fila es igual a cero.*

Camino orientado: Es una sucesión de arcos $e_1 e_2 \dots e_k$ tal que el primer elemento del arco e_i coincide con el segundo de e_{i-1} y el segundo elemento de e_i con el primero de e_{i+1} con $i = 2, \dots, k-1$

Grafo fuertemente conexo: Es un digrafo G tal que para todo par de vértices $u, v \in V_G$ existe un camino orientado de u a v.

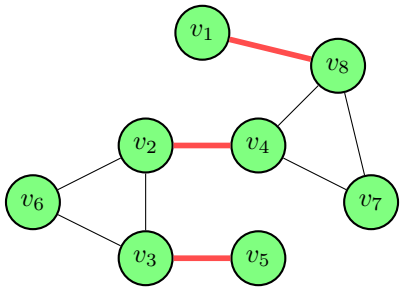
3. Árboles

Árbol: Grafo conexo sin circuitos simples.



Árbol trivial Árbol T

Puente: Una arista e de un grafo G tal que $G - e$ tiene más componentes conexas que G.



Puentes: $(v_1, v_8), (v_2, v_4), (v_3, v_5)$

Lema 1. *La unión de dos caminos simples distintos entre dos vértices contiene un circuito simple.*

Lema 2. *Sea $G = (V, X)$ un grafo conexo y $e \in X$. $G - e = (V, X \backslash \{e\})$ es conexo si y solo si e pertenece a un circuito simple de G.*

En otras palabras: Una arista $e \in X$ es puente si y solo si e no pertence a ningún circuito simple de G

DEMOSTRACIÓN

$\Rightarrow$) Sea $e = (u, v) \in X$. Por hipotesis $G - e$ es conexo. Entonces existe un camino simple P_{uw} entre u y w en $G - e$ (que no usa a e). Luego podemos definir a $C = P_{uw} + e$ como un circuito simple de G que contiene a e.

$\Leftarrow$) Sea C un circuito simple de G que contiene a $e = (u, w)$. Entonces podemos partir a C en la arista e y un camino simple P_{uw}) entre u y w (que no contiene a e).

Como G es conexo, hay camino entre todo par de vértices. Si esos caminos no una a e, entonces siguen estando en $G - e$.

Si un camino de G usa e, en $G - e$ hay un camino alternativo que es cambiando a e por P_{uw} en Q. Entonces sigue habiendo camino entre todo par de vértices en $G - e$. Osea $G - e$ es conexo.

Teorema 3. *Dado un grafo $G = (V, X)$, son equivalentes:*

1. *G es un árbol.*
2. *G es un grafo sin circuitos simples y para toda arista e tal que $e \notin X$, $G + e = (V, X \cup \{e\})$ tiene exactamente un circuito simple. Además, ese circuito contiene a e.*
3. *Existe exactamente un camino simple entre todo par de vértices.*
4. *G es conexo pero si se quita cualquier arista a G queda un grafo no conexo (toda arista es puente).*

DEMOSTRACIÓN

Vamos a demostrar $1 \Rightarrow 2$, $2 \Rightarrow 3$, $3 \Rightarrow 4$ y $4 \Rightarrow 1$.

1 $\Rightarrow$ 2) G es árbol, es decir, conexo y sin circuitos. Sea $e = (u, w) \notin X$. Como G es conexo existe un camino simple P_{uw} en G entre u y w. Entonces $P_{uw} + e$ es un circuito simple de $G + e$.
Ahora nos falta ver que no se puede haber generado más de un circuito simple. Vamos a hacerlo por el absurdo. Como sabemos que antes de agregar e, G no tenía circuitos, todo ciruito simple de $G + e$ tiene que contener a e, ya que de lo contrario ese circuito estaría en G.
Entonces supongamos que existen dos circuitos simples C y C' en $G + e$ que contienen e. Podemos partir estos circuitos en

$$C = P_{uw} + e$$

$$C' = P'_{uw} + e$$

P_{uw} y P'_{uw} son dos caminos simples distintos entre u y v en G. Por el Lema **??**, la unión de estos dos cáminos en G es un circuito simple. Esto es absurdo ya que G es un árbol.

2 $\Rightarrow$ 3) Primero veamos que G) es conexo, es decir que hay camino entre todo par de vértices u y v. Si $(u, w) \in X_G$, el camino es la arista (u, w). Si u y v no son adyacentes en G, por hipotesis, $G + (u, v)$ contiene exactamente un circuito simple C que contiene a u, w. Podemos partir ese circuito en $C = P_{uw} + (u, w)$ y P_{uw} es un camino simple entre u, v.

3 ⇒ 4) Por hipotesis, G es conexo. Supongamos que existe $e = (u, v) \in X$ tal que $G - e$ es conexo. Por Lema **??**, e pertenece a un circuito simple C de G. Partir a $C = P_{u,v} + (u, v)$, entonces $P_{uv}]$ y (u, v) son dos caminos simples distintos entre u y v, contradiciento 3.
4 ⇒ 1) Por hipotesis, G es conexo. Por contradicción, supongamos que no es un árbol, osea que tiene un circuito C. Sea $e = (u, w))$ una arista de ese circuito. Por Lema **??**, $G - e$ es conexo, pero esto contradice 4. Luego G es conexo y sin circuitos, osea es un árbol.

Hoja: Una hoja es un nodo de grado 1.

Lema 3. *Todo árbol no trivial T (de al menos dos vértices) tiene al menos dos hojas.*

DEMOSTRACIÓN

Sea $P : v_1 \dots v_k$ un camino simple máximal (no extendible por sus extremos) en el árbol T. Veamos que v_1) y v_k son hojas.
Supongamos que $d(v_1) > 1$. Entonces v_1 tiene otro vértice adyacente w además de v_2 y $w \neq v_2$. Como P es máximal, w necesariamente ya tiene que estar en P (sino podriamos agregarlo y seguir extiendiendo el camino). Luego, $P_{v_1,w} + (v_1, w)$ es un circuito, contradiciento que T es un árbol. Se puede utilizar el mismo razonamiento para deducir que $d(v_k) = 1$.

Lema 4. *Sea $G = (V, X)$ un árbol. Entonces $m = n - 1$.*

DEMOSTRACIÓN

Vamos a hacer inducción en la cantidad de vértices.

Caso base ($n = 1$): Se cumple por que $n = 1$ y $m = 0$

Caso inductivo: Sea $T = (V, X)$ un árbol con k vértices ($k > 1$). Nuestra hipotesis inductiva es:

$$\forall\ T' \ /\ n_{T'} \leq k \implies m_{T'} = n_{T'} - 1$$

Por lema **??**, T tiene al menos una hoja u. Definamos $T - u = (V \backslash \{u\},\ X \backslash \{(u, v)\}$. $T - u$ es conexo y no tiene circuitos, osea que es un árbol de $k - 1$. Por hipotesis inductiva, $T - u$ tiene $k - 2$ aristas. Como $d(u) = 1$, T tiene una arista más. Luego T tiene $k - 1$ aristas.

Bosque: Es un grafo sin circuitos simples.

Corolario 2. *Sea $G = (V, X)$ sin circuitos simples y c componentes conexas. Entonces $m = n - c$.*

DEMOSTRACIÓN

Para $i = 1, \ldots, c$, sea n_i la cantidad de vértices de la componente i y m_i la cantidad de aristas.
Como cada componente conexa de G es un árbol, podemos aplicar el lema **??**. Entonces $m_i = n_i - 1 \; \forall \; i = 1 \ldots c$. Sumando, obtenemos que

$$m = \sum_{i=1}^{c} m_i = sum_{i=1}^{c} n_i - 1 = n - c$$

Corolario 3. *Sean $G = (V, X)$ un bosque con c componentes conexas. Entonces $m \geq n - c$*

DEMOSTRACIÓN

Si G tiene circuitos, removerlos sacando una arista por vez hasta que el grafo resultante $\hat{G}$ sea sin circuitos. Entonces, por Lema **??**, $\hat{m} = n - \hat{c}$

Teorema 4. *Dado un grafo G son equivalentes:*

1. *G es un árbol*
2. *G es un grafo sin circuitos simples y $m = n - 1$*
3. *G es conexo y $m = n - 1$*

DEMOSTRACIÓN

1 ⇒ 2) Como G es árbol, G no tiene circuitos y $m = n - 1$ por Lema **??**

2 ⇒ 3) Sea c lacantidad de componentes conexas de G. Por corolario **??** , $n - 1 = m = n - c$.

3 ⇒ 1) Por contradicción, supongamos que G tiene un circuito simple. Sea $e = (v, w)$ una arista del circuito. Entonces, por Lema **??**, $G' = G - e$ es conexo y $m' = n - 2$, contradiciendo el corolario **??**. Luego, G no puede tener circuitos simples. G es árbol.

3.1. Árboles enraizados

Un **árbol enraizado** es un árbol qeue tiene un vértice distinguido que llamamos **raíz**. Explicitamente queda definido un árbol dirigido, considerando caminos orientados desde la raíz al resto de los vértices.

- Los vértices **internos** de un árbol son aquellos que no son ni hojas ni la raíz.
- El **nivel** de un vértice de un árbol con raíz en la distancia de la raíz a ese vértice.

- Decimos que dos vértices adyacentes tienen **relación parde-hijo**, siendo el padre el vértice de menor nivel.
- La **altura** h de un árbol con raíz es la distancia desde la raíz al vértice más lejano.
- Un árbol se dice (exactamente) $m-ario$ si todos sus vértices,salvo las hojas y las raíz tienen grado (exactamente) a lo sumo $m+1$ y la raíz (exactamanete) a lo sumo m.
- Un árbol se dice **balanceado** si todas sus hojas están a nivel h ó $h-1$.
- Un árbol se dice **balanceado completo** si todas sus hojas están a nivel h.

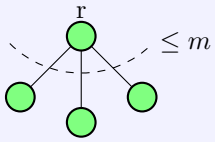
Teorema 5. *Sea T un árbol m-ario de altura h con l hojas entonces:*

1. *T tiene a lo sumo m^h hojas.*
2. *$h \geq \lceil \log_m l \rceil$*
3. *Si T es un árbol exáctamente m-ario balancaeando completo entonces $h = \lceil \log_m l \rceil$*

DEMOSTRACIÓN DE 1

Lo hacemos por inducción en la altura del árbol:

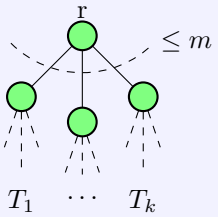
Caso base ($h = 1$): Por definición, en un árbol m, cada nodo tiene a lo sumo m hijos. Si tomamos la raíz, entonces $d(r) \leq m$. Y como el árbol es de altura 1, todos sus hijos son hojas entonces: $\#hojas = d(r) \leq m = m^1$



Paso inductivo: Sea $T = (V, X)$ un árbol m-ario de altura $h > 1$. Nuestra hipotesis inductiva es:

Todo árbol m-ario de altura $h' < h$ tiene a lo sumo $m^{h'}$ hojas.

Sea r la raíz de T, como T es un árbol m-ario, $d(r) = k \leq m$. Sean $T_1, \ldots, T_k$ las componentes conexas de $T - r$:



Cada T_i tiene altura $h_i \leq h-1$, osea que por la HI vale que $\#hojas(T_i) \leq m^{h_i} \leq m^{h-1}$. Además, de que todas las hojas de cada T_i son hojas T.
Si hay algun T_i cuya altura sea 0, entonces no tiene hojas pero T_i era un hoja de T, sin embargo sigue valiendo que $0 \leq m^0 \leq m^{h-1}$.
Entonces:

$$\begin{aligned}\#hojas(T) &= \sum_{\substack{i=0 \\ h_i>0}}^{k} \underbrace{\#hojas(T_i)}_{\leq m^{h_i}} + \sum_{\substack{i=0 \\ h_i=0}}^{k} \underbrace{1}_{\leq m^{h_1}} \\ &\leq \sum_{i=0}^{k} m^{h_i} \leq \sum_{i=0}^{k} m^{h-1} \leq \sum_{i=0}^{m} m^{h-1} \\ &\leq m \times m^{h-1} \leq m^h\end{aligned}$$

DEMOSTRACIÓN DE 2

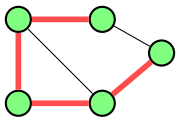
Por 1, sabemos que $l \leq m^h \implies \log_m l \leq h$ y como h es entero $\lceil \log_m l \rceil \leq h$

DEMOSTRACIÓN DE 3

Si T es un árbol exactamente m-ario balanceado completo, entonces los $T_1, \ldots, T_k$ de la demostración del punto 1 son también árboles m-arios balanceados completos (por lo que cumplen la hipotesis inductiva) y los pasos son todas igualdades.

3.2. Árboles generadores

Un **árbol generador** (AG de un grafo G es un subgrafo generador (que tiene el mismo conjunto de vértices) de G que es árbol.



Grafo G y un AG(G) T

Teorema 6. *Sea $G = (V, X)$ un grafo conexo:*

1. *G tiene (al menos) un árbol generador.*
2. *G tiene un único árbol generador si y solo si G es un árbol.*
3. *Sea $T = (V, X_T)$ y $e \in X \backslash X_T$. Sea $f \neq e$ una arista del circuito que se genera cuando se agrega e a T. Entonces $T + e - f$ es un árbol generador de G.*

DEMOSTRACIÓN DE 1

Sea $G = (V, X)$ un grafo conexo, podemos construir un árbol generador T de la siguiente manera:

Definimos $T = (V, X_T)$ con $X_T = X$.

while T tenga algún circuito **do**

Seleccionamos $e \in X_T$ que pertenezca a un circuito de T.

$T = T - e$

end while

Como todas las aristas removidas pertenecen a un circuito (no son puentes), cuando termina el procedimiento T es un subgrafo conexo generador de G (por lema **??**) y todas sus aristas son puente. Entonces, por el teorema **??**, T es un árbol.

DEMOSTRACIÓN DE 2

$\Rightarrow$) G tiene un único árbol generador. Supongamos que no es un árbol, entonces tiene por lo menos un circuito simple C (supongamos que tiene exactamente uno, sin perdida de generalidad).
Sean e y f dos aristas de C, entonces siguiendo el procedimiento de la demostración anterior podemos formar dos árboles generadores distintos de G: $T_1 = G - e$ y $T_2 = G - f$, si elegimos sacar e o f, respectivamente. En ambos casos, sabemos que son árboles porque C era el único ciclo simple de G y quitarle una arista lo rompe. Luego, llegamos a un absurdo que proviene de suponer que G no era un árbol.
$\Leftarrow$ G es un árbol.Por teorema **??**, todas sus aristas son puente, es decir que cualquier arista que e saquemos hace que $G - e$ deje de ser conexo. Luego, G es un árbol generador de si mismo.

DEMOSTRACIÓN DE 3

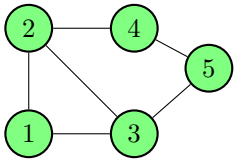
Como T es un árbol generador de G, entonces por el teorema **??**, sabemos que $T + e$ tiene exactamente un circuito C. Sea $e \neq f$ una arista de C, por lema **??**, si se quita una arista de un circuito el grafo sigue siendo conexo, es decir $T + e - f$ es conexo.
Entonces $T + e - f$ es un grafo generador G, conexo y con $n - 1$ aristas, lo que implica que $T + e - f$ es árbol generador de G.

3.3. Recorrido de árboles o grafos

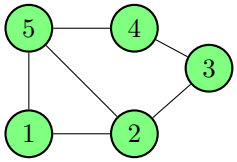
Hay dos formas de recorrer un grafo: **Breadth-First Search (BFS)** o **Depth-First Search (DFS)**.

En el BFS, se comienza por el nivel 0 (la raíz) y se visita cada vértice en un nivel antes de pasar al siguiente.

En el DFS, se comienza por la raíz y se explora cada rama lo más profundo posible antes de retroceder.



Orden de recorrido BFS



Orden de recorrido DFS

Para ambos tipos de recorrido el algoritmo es similar, se diferencian en las estructuras que se usan para implementarlos:

```
procedure BFS(G = (V, X))
    r ← v ∈ V
    to_visit ← {r}
    r.used ← true
    while ¬to_visit.empty() do
        i ← to_visit.pop()
        for ∀ (i, j) ∈ X / ¬j.used do
            j.used ← true
            to_visite.push(j)
        end for
    end while
end procedure
```

Para implementar un **BFS** un hay que utilizar una cola (`Queue`), para un **DFS**, una cola (**Stack**).

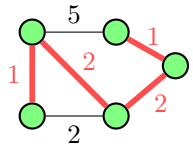
3.4. Árbol generador mínimo

Grafo pesado: Grafo que tiene un costo asociado a sus aristas o vértices.

Sea $T = (V, X)$ un árbol y $l : X \to \mathbb{R}$ una función que asigna costos a las aristas de T. Se define el **costo** de T como $l(T) = \sum_{e \in T} l(e)$.

Dado un grafo $G = (V, X)$, un **árbol generador mínimo** de G, $AGM(G) = T$, es un árbol generador de G de mínimo costo, es decir:

$$l(T) \leq l(T') \ \forall \ T' \text{ árbol generador de} G$$



Grafo pesado G y un AGM(G)

Dado un grafo pesado en las aristas $G = (V, X)$, el problema de árbol generador mínimo consiste en encontrar un AGM de G.

3.4.1. Algoritmo de Prim

El algoritmo de Prim es un algoritmo **goloso** que construye incrementalmente dos conjuntos, uno de vértices V_T y uno de aristas X_T que comienza vacío. En cada iteración se agrega un elemento a cada uno de estos conjuntos. Cuanto $V_T = V$ el algoritmo termina y las aristas de X_T definen un AGM de G.

En cada paso, se selecciona la arista de menor costo entre las que tiene un extremo en V_T y el otro en $V \backslash V_T$. Esta arista es agregada a X_T y el extremo a V_T.

procedure PRIM($\mathtt{G} = (\mathtt{V}, \mathtt{X})$)
  $\mathtt{V_T} \leftarrow \{\mathtt{u}\}$ ▷ Comenzamos con cualquier vértice de G
  $\mathtt{X_T} \leftarrow \emptyset$
  $\mathtt{i} \leftarrow 1$
  while $\mathtt{i} \leq \mathtt{n} - 1$ **do**
    $\mathtt{e} \leftarrow \mathtt{argmin}\{\mathtt{l(e)}, \mathtt{e} = (\mathtt{u}, \mathtt{w}),\ \mathtt{u} \in \mathtt{V_T} \wedge \mathtt{w} \in \mathtt{V} \backslash \mathtt{V_T}\}$
    $\mathtt{X_T} \leftarrow \mathtt{X_T} \cup \{\mathtt{e}\}$
    $\mathtt{V_T} \leftarrow \mathtt{V_T} \cup \{\mathtt{w}\}$
    $\mathtt{i} \leftarrow \mathtt{i} + 1$
  end while
  return $\mathtt{T} = (\mathtt{V_T}, \mathtt{X_T})$
end procedure

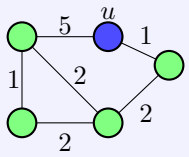
Notaremos $T_k = (V_k, X_k)$ al grafo que el algoritmo de Prim construyó al finalizar la iteración k, para $0 \leq k \leq n - 1$. $T_0 = (V_0, X_0)$ se refiere a la inicialización antes de entrar a la primera iteración.

Proposición 7. *Dado $G = (V, X)$ un grafo conexo. $T_k = (V_k, X_k)$, $0 \leq k \leq n - 1$, es árbol y subgrafo de árbol generador mínimo de G.*

DEMOSTRACIÓN

Lo hacemos por inducción en las iteraciones del ciclo:

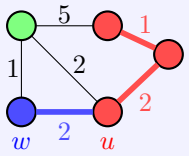
Caso base (k = 0): Antes de ingresar al ciclo, $T_0 = (\{u\}, \emptyset)$ es árbol y subgrafo de todo AGM de G.



Paso T_0 de Prim sobre G

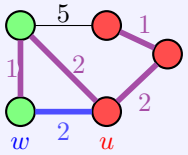
Paso inductivo: Consideremos T_k con $k > 1$, nuestra hipotesis inductiva es: $\forall\ k' < k,\ T'_k$ es árbol y subgrafo de algún AGM $T = (V, X_T)$ de G.

Llamemos w al vertice agregado en la iteración k y e a la arista. Es decir $T_k = (V_k, X_k)$ con $V_k = V_{k-1} \cup \{w\}$ y $X_k = X_{x-1} \cup \{e = (u, w)\}$, $u \in V_{k-1}$ y $w \notin V_{k-1}$.



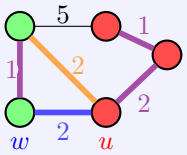
Paso T_k de Prim sobre G

- T_k **es un árbol:** Por HI, T_{k-1} es árbol. Como T_k tiene un vértice y un arista más que T_{k-1} y es conexo entonces T_k es árbol.
- T_k **es subgrafo de un AGM:** Sea $T = (V, X_T)$ un AGM tal que T_{k-1} es súbgrafo T (T existe por HI):
  - Si $e \in X_T$ entonces T_k también es subgrafo de T.
  - Si $e \notin X_T$, veamos que podemos armar un AGM a partir de T que contenga T_{k-1} y a e:



Arbol T tal que $e \notin X_T$

Como T es un árbol, $T + e$ tiene un circuito simple C que contiene a e (por teorema **??**). Este circuito está formado por el único camino P_{uw} entre u y w que tiene T más la arista e. Sea $f \in X_T$ la primer arista de P_{uw} tal que tiene un extremo em T_{k-1} y el otro no (existe, porque el camino empieza en u que esta dentro de T_{k-1} y termina en w que no lo está.



Definimos $T' = T + e - f$, veamos que es AGM:

- T' es un árbol generador de G por el teorema **??**
- T_k es subgrafo de T'
- T' es AGM de G: Como f era una arista elegible al comienzo de la iteración k, pero el algoritmo eligió e, seguro se cumple $l(e) \leq l(f)$. Entonces

$$l(T') = l(T) + l(e) - l(f) \leq T$$

Entonces T' es un AGM y además T_k es un subgrafo de T'.

Teorema 7. *El algoritmo de Prim es correcto, es decir dado un grafo G conexo determina un árbol generador mínimo de G.*

DEMOSTRACIÓN

Al analizar la iteración $n-1$, por la propocisión **??**, T_{n-1} es un árbol y subgrafo de algún T AGM de G.
Además, T_{n-1} es subgrafo generador de G) ya que cada iteración del algoritmo agrega un vértice distinto a V_T y, entonces, $V_{n-1} = V$.
Entonces $T_{n-1} = T$, AGM de G

3.4.2. Algoritmo de Kruskal

Este algoritmo ordena las aristas del grafo de forma creciente según su peso y en cada paso elige la siguiente arista que no forme circuito con las aristas ya elegidas. También es un algoritmo goloso.

procedure KURSKAL($\mathtt{G} = (\mathtt{V}, \mathtt{X})$)
  $\mathtt{X_T} \leftarrow \emptyset$
  $\mathtt{i} \leftarrow 1$
  while $\mathtt{i} \leq \mathtt{n} - 1$ **do**
    $\mathtt{e} \leftarrow \mathtt{argmin}\{\mathtt{l(e)},\ \mathtt{e\ no\ forma\ circuito\ con\ las\ aristas\ de\ X_T}\}$
    $\mathtt{X_T} \leftarrow \mathtt{X_T} \cup \{\mathtt{e}\}$
    $\mathtt{i} \leftarrow \mathtt{i} + 1$
  end while
  return $\mathtt{T} = (\mathtt{V}, \mathtt{X_T})$
end procedure

Al comenzar el algoritmo, cuando todavía no se seleccionó arista alguna, cada vértice del grafo forma una componente conexa distinta (es un bosque de árboles triviales). En cada iteración, se elige una arista que tiene extremos en dos componentes conexas distintas del grafo obtenido en

el paso anterior, convertindiose en nuevo bosque. El algoritmo termina cuando el bosque pasa a ser un árbol, es decir, se vuelve conexo.

La demostración de correctitud de Kruskal es similar a la de Prim: En cada iteración, el grafo que arma el algoritmo es un bosque. Al final la última iteración, como se incorporaron $n-1$ aristas, pasa a ser árbol (sin circuitos y con $m-1$ aristas).

En este caso, debemos demostrar que vale el invariante:

Proposición 8. *Sea T_k el grafo generado en la k-ésima iteración, $T_k = (V_k, X_k)$, $0 \leq k \leq n-1$, es un bosque y subgrafo de árbol generador mínimo de G.*

DEMOSTRACIÓN

La demostración es muy similar a la de la proposición **??**. Si $e = (u, w)$ es la arista incorporada en el k-ésimo lugar, la única diferencia significativa es la definición de la arista f en el paso inductivo cuando $e \notin T$ (AGM(G) tal que $T_{k-1} \in T$).
En este caso, debemos elegir una f que pertenezca al circuito $C = P_{uw}+e$ que tenga un extremo en la componente conexa a la que pertenece u y el otro fuera de esa compenente.

4. Caminos mínimos

Los grafos (pesados o no) son la estructura natural para modelar redes en las cuales uno quiere ir de un punto a otro de la red atravesando una secuencia de enlaces. Formalmente, sea $G = (V, X)$ un grafo y $l : X \to \mathbb{R}$ una función de longitud/peso para las aristas de G:

- La **longitud** de un camino C entre dos vértices v y w es la suma de las longitudes de las aristas del camino:
$$l(C) = \sum_{e \in C} l(e)$$
- Un **camino mínimo** C^0 entre v y w es un caminot entre v y w tal que
$$l(C^0) = \text{mín}\{l(C) | C \text{es un camino entre } v \text{ y } w\}$$

Dado un grafo G, se pueden definir tres variantes de problemas sobre caminos mínimos:

- **Único origen - único destino:** Determinar un camino mínimo entre dos vértices específicos v y w.
- **Único origen - múltiples destinos:** Determinar un camino mínimo desde un vértice específico v al resto de los vértices de G.
- **Múltiples origenes - múltiples destinos:** Determinar una camino mínimo entre todos los vértices de G.

Generalmente, los algoritmos para resolver problemas de camino mínimo se basan en que todo subcamino de un camino mínimo entre dos vértices es un camino mínimo.

Proposición 9. *Dado un grafo $G = (V, X)$ con una función de peso $l : X \to \mathbb{R}$, sea $P : v_1 \dots v_k$ un camino mínimo de v_1 a v_k. Entonces $\forall\ 1 \leq i \leq j \leq k$, $P_{v_i v_j}$ es un camino mínimo desde v_i a v_j.*

DEMOSTRACIÓN

Podemos descomponer al camino P en $P_{v_1 v_i} + P_{v_i v_j} + P_{v_j v_k}$, entonces $l(P) = l(P_{v_1 v_i}) + l(P_{v_i v_j})) + l(P_{v_j v_k})$.
Por el absurdo, asumamos que $P_{v_i v_j}$ no es un camino mínimo desde v_i a v_j. Llamemos P' a un camino entre estos dos vértices, entonces $l(P') \leq P_{v_i v_j}$. Entonces podemos armar $P'' = P_{v_1 v_i} + P' + P_{v_j v_k}$ y

$$l(P'') = l(P_{v_1 v_i}) + l(P') + l(P_{v_j v_k}) < l(P_{v_1 v_i}) + l(P_{v_i v_j})) + l(P_{v_j v_k}) = l(P)$$

Luego P no es un camino mínimo entre v y w.

Dos consideraciones a tener en cuenta:

- **Aristas con peso negativo:** Si el digrafo G no contiene ciclos de peso negativo o contiene alguno pero no es alcanzable desde v, entonces el problema sigue estando bien definido, aunque algunos caminos puedan tener longitud negativa. Sin embargo, si G tiene algún circuito de peso negativo alcanzable desde v, el concepto de camino mínimo deja de estar bien definido: Si w pertenece a un camino con ciclo de peso negativo, ningún camino de v a w puede ser mínimo porque siempre se puede obtener un camino de peso menor siguiendo ese camino pero atravesando el ciclo de peso negativo una vez más.
- **Circuitos:** Siempre existe un camino mínimo que no contiene circuitos (si el problema está bien definido):
  - Sabemos que no puede tener un circuito negativo porque si no no está bien definido.
  - Si tiene un circuito de peso positivo, entonces podemos sacarlo obteniendo un camino con el mismo origen y destino de menor longitud (osea que no era camino mínimo).
  - Si tiene un circuito de peso cero, entonces sacandolo obtenemos un camino sin cirtutos del mismo peso.

4.1. Camino mínimo con un único origen y múltiples destinos

Dado $G = (V, X)$ un digrafo, $l : X \to \mathbb{R}$ una función que asigna a cada arco una cierta longitud y $v \in V$ un vértice del digrafo, queremos calcular los caminos mínimos desde v al resto de los vértices.

4.1.1. BFS

En el caso de que todas las aristas tengan igual longitud, este problema se traduce en encontrar los caminos que definan las distancias (de menor cantidad de aristas). Para esto se puede adaptar el BFS (Sección **??**):

Lema 5. *Dado $G = (V, X)$ un digrafo y $v \in V$. Sea $\texttt{to\_visit}_k = [v_1, \ldots, v_r]$ la cola de nodos que se obtiene al finalizar la iteración k del algoritmo. Se cumple que todos los nodo de la misma están a distancia $\texttt{dist}[v_1]$ o $\texttt{dist}[v_1] + 1$ de v, formalmente:*

```
procedure BFS(G = (V, X), v ∈ V)
    pred[w] = antecesor de w en un camino minimo desde v
    dist[w] = distancia desde v a w
    pred[v] ← 0
    dist[v] ← 0
    to_visit ← {v}
    for w ∈ V\{v} do
        dist[w] ← ∞
    end for
    while ¬to_visit.empty() do
        x ← to_visit.pop()
```

```
        for w tal que (x → w) ∈ X / ∧ dist[w] = ∞ do
            pred[w] ← x
            dist[w] ← dist[x] + 1
            to_visite.push(w)
        end for
    end while
    return [pred, dist]
end procedure
```

- $dist[v_1] + 1 \geq dist[v_r]$ *y*
- $dist[v_i] \leq dist[v_{i+1}]$ *para todo* $i = 1, \dots r - 1$

DEMOSTRACIÓN

Lo hacemos por inducción en las iteraciones del algoritmo.

Caso base: Antes de entrar al ciclo, la cola solo tiene v y se asigna $\texttt{dist}[v] = 0$. Como v es el único elemento de la cola, se cumple la propiedad.

Paso inductivo: Consideremos la iteración k. Llamemos $\texttt{to\_visit}_k$ con $k \geq 1$ al valor de la variable `to_visit` al finalizar la iteración k del algoritmo.
Nuestra hipotesis inductiva es: Para todo $k' < k$, vale que $\texttt{to\_visit}_{k'} = [v_1^{k'}, \dots, v_{r_k'}^{k'}]$ cumple que $\texttt{dist}[v_1]+1 \geq \texttt{dist}[v_r]$ y $\texttt{dist}[v_i^{k'}] \leq \texttt{dist}[v_{i+1}^{k'}]$ para todo $i = 1, \dots r_{k'} - 1$.

- Si $\texttt{to\_visit}_{k-1}$ tiene un solo elemento (el que se desencola), como todos los vértices que ingresan tienen igual valor en `dist`, se cumple la propiedad.
- Si $\texttt{to\_visit}_{k-1} = [u_1, u_2, \dots, u_r]$ tiene más de un elemento, sea $w_1, \dots, w_s$ los vértices que se encolan en la iteración k. Entonces veamos que $\texttt{to\_visit}_k = [u_2, \dots, z, w_1, \dots, w_s]$ cumple la propiedad:
  - Como w_s fue agregada en este paso del algoritmo, sabemos que $\texttt{dist}[w_s] = \texttt{dist}[u_1] + 1$. Por hipotesis inductiva sabemos que $\texttt{dist}[u_1] \leq \texttt{dist}[u_2]$

    $$\begin{aligned} \texttt{dist}[u_1] \leq \texttt{dist}[u_2] &\implies \texttt{dist}[u_1] + 1 \leq \texttt{dist}[u_2] + 1 \\ &\implies \texttt{dist}[w_s] \leq \texttt{dist}[u_2] + 1 \end{aligned}$$

    Entonces la primer parte del lema se cumple.

- Todos los nodos encolados en este paso están a $\texttt{dist}[u_1] + 1$ de v. Osea que $\texttt{dist}[w_i] = \texttt{dist}[u_1] + 1 \ \forall \ i = 1 \dots s$, entonces tambien vale que $\texttt{dist}[w_i] \leq \texttt{dist}[w_{i+1}]$.

  Por hipotesis inductiva sabemos que $\texttt{dist}[u_i] \leq \texttt{dist}[u_{i+1}]$ para todo $i = 1 \dots r$.

  Por hipotesis inductiva sabemos $\texttt{dist}[u_1]+1 \geq \texttt{dist}[u_r]$, luego $\texttt{dist}[w_1] \geq \texttt{dist}[u_r]$.

  Luego se cumple la segunda parte de la propiedad.

■

Corolario 4. *Dago $G = (V, X)$ un digrafo y $v, w \in V$. Si el vértice w ingresa a `to_visit` antes que el vértice v, entonces se cumple que $dist[w] \leq dist[v]$ durante toda la corrida del algoritmo.*

Lema 6. *Dado $G = (V, X)$ un digrafo y $v \in V$. Se cumple que $\texttt{dist}[w] \geq d(v, w)$ para todo $w \in V$ en todo momento del algoritmo.*

DEMOSTRACIÓN

Haremos inducción en las iteraciones de k.

Caso base: Para $k = 0$ (antes de entrar al ciclo), $\texttt{to\_visit}_0 = [v]$, $\texttt{dist}[v] = 0$ y $\texttt{dist}[w] = \infty$ para todo $w \in V$ tal que $w \neq v$.
Como $d(v, v) = 0$, se cumple que $\texttt{dist}[w] \geq d(v, w)$ para todo $w \in V$.

Paso inductivo: Nuestra hipotesis inductiva es: $\texttt{dist}_{k-1}[w] \geq d(v, w) \ \forall \ w \in V$ donde $\texttt{dist}_{k-1}$ es el valor de la variable `dist` al finalizar la iteración $k - 1$.
Sea x el vértice que se desencolaen la iteración k y $x_1, \dots, x_s$ los vértices que se encolan. La variable `dist` solo se modifica en las posiciones de $x_1, \dots, x_s$. Es decir que $\text{dist}_k[w] = \texttt{dist}_{k-1}[w]$ para todo $w \in V/\{x_1, \dots, x_s\}$, entonces por hipotesis inductiva vale $\texttt{dist}_k[w] \geq d(v, w)$ para $w \in V/\{x_1, \dots, x_s\}$.
Para las posiciones modificadas, tenemos que

$$\texttt{dist}[x_i] = \texttt{dist}_{k-1}[x] + 1$$

Por hipotesis inductiva sabemos que $\texttt{dist}_{k-1}[x] \geq d(v, x)$, además como x_i fue agregado a la cola, sabemos que existe $(x, x_i) \in V$.
Sea $P_i = P_{vx} + (x, x_i)$ un camino entre v y x_i. Si P_{vx} es un camino mínimo entre v y x entonces

$$l(P_i) = l(P_{vx}) + 1 = d(v, x) + 1 \underset{HI}{\leq} \texttt{dist}_{k-1}[x] + 1 = \texttt{dist}[w_i]$$

Como $d(v, x_1)$ es la longitud de los caminos mínimos entre v y x_i, sabemos que $d(v, x_i) \leq l(P_i)$, entonces $d(v, x_1) \leq \texttt{dist}[w_i]$.

Teorema 8. *Dado $G = (V, X)$ un digrafo y $v \in V$. El algoritmo BFS enunciado calcula $d(v, w)$ para todo $w \in V$*

DEMOSTRACIÓN

Por el lema **??**, sabemos que $\texttt{dist}[w] \geq d(v, w)$ para todo $w \in V$. Supongamos ahora que para algún vértice no se cumple la igualdad.
Sea $x \in V$ el vértice con menor $d(v, x)$ tal que $\texttt{dist}[x] > d(v, x)$:

- Sabemos que $v \neq x$ porque $\texttt{dist}[v] = d(v, v) = 0$ (este valor se asigna antes de entrar a la primera iteración).
- Sea $P = v, \ldots, y, x$ un camino mínimo desde v a x, entonces

  $$d(v, x) = l(P) = d(v, y) + 1$$

  (el subcamino de v a y es camino mínimo por la proposicion **??**).

  Ahora, como x es el vértice de menor distancia a v tal que $\texttt{dist}[x] > d(v, x)$ e y está a menor distancia de v que w entonces vale que $\texttt{dist}[y] = d(v, y)$.

  Entonces, en la iteración del algoritmo que analizamos y, tiene que pasar alguna de las situaciones siguientes:

  1. x ya fue desencolado de `to_visit`
  2. x ya está en la cola
  3. x todavía no está en la cola

  Si ocurren la situacion **??** y **??**,como x fue encolado antes que y, vale que $\texttt{dist}[x] \leq \texttt{dist}[y]$, entonces:

  $$d(v, x) = \texttt{dist}[y] + 1 \geq dist[x] + 1 > dist[x]$$

  Esto genera un absurdo.

  Si ocurre la situacion **??**, x todavía no está en la cola entonces se lo agrega porque existe $(y, x) \in X$. Además, el algoritmo fijará $\texttt{dist}[x] = \texttt{dist}[y] + 1 = d(v, x)$, contradiciendo nuestra supocisión.

4.1.2. Algoritmo de Dijkstra

El algoritmo de Dijsktra es un algoritmo goloso que construye un árbol de caminos mínimos con raiz en v. En cada iteración agrega el vértice más cercano a v de entre todos los que todavía no fueron agregados al árbol.

Este algoritmo asume que las longitudes de arco son positivas. El grafo puede ser orientado o no.

procedure DIJSKTRA($G = (V, X),\ v \in V$)
  π[u] = valor del camino mínimo desde v a u
  S $\leftarrow$ {v}
  π[v] $\leftarrow$ 0 ▷ Inicializamos π y *textttpred*
  pred[v] $\leftarrow$ 0
  for $u \in V$ **do**
    if (v, u) $\in$ X **then**
      π[u] = l(v, u)
      pred[u] = v
    else
      π[u] = ∞
      pred[u] = ∞
    end if
  end for
  while S $\neq$ V **do** ▷ Calculamos los caminos mínimos
    w $\leftarrow$ **arg** mín{π[u], u $\in$ V\S}
    S $\leftarrow$ S $\cup$ {w}
    for u $\in$ V\S $\wedge$ (w, u) $\in$ X **do**
      if π[u] > π[w] + l(w, u) **then**
        π[u] $\leftarrow$ π[w] + l(w, u)
        pred[u] = w
      end if
    end for
  end while
  return [π, *pred*]
end procedure

Lema 7. *Dado un digrafo $G = (V, X)$ con pesos positivos en los arcos, al finalizar la iteración k el algoritmo de Dijsktra determina, siguiendo hacia atrás* `pred` *hasta llegar a v, un caminom mínimo entre el vértice v y cada vértice $u \in S_k$, con longitud π[u].*

DEMOSTRACIÓN

Haremos inducción en la cantidad de iteraciones.

Caso base ($k = 0$): Antes de entrar por primera vez al ciclo, vale $\mathtt{S}_0 = \{v\}$ y $\pi[v] = 0$. Esto es correcto ya que el camíno mínimo desde v hasta v es 0 por definición.

Paso inductivo: Consideremos la iteración k con $k \geq 1$.
Nuestra hipotesis inductiva es: Al terminar la iteración k' ($k' < k$), $\forall\ u \in \mathtt{S}_{k'}$, $\pi[u]$ es la longitud de un camino mínimo de v a u finalizado en $\mathtt{pred}[u]$.
Sea u el vértice agregado a $\mathtt{S}$ en la iteración k. $\mathtt{S}_k = \mathtt{S}_{k-1} \cup \{u\}$ con $\mathtt{pred}[u] = w$. Como $w \in \mathtt{S}_{k-1}$, por hipotesis inductiva, $\pi[w]$ es el válor del camino mínimo desde v a w definido por $\mathtt{pred}$. Llamemos P_{vw} a ese camino mínimo.
Sea $P = P_{vw} + (w, u)$ y P' otro camino de v a u y sea $y \in P'$ el primer vértice que no está en $\mathtt{S}_{k-1}$ (debe existir porque $v \in \mathtt{S}_{k-1}$ y $u \notin \mathtt{S}_{k-1}$):

- Si $y = u$, sea x el predecesor de u (que está en $\mathtt{S}_k$) entonces $l(P') = \pi[x] + l(x, u)$

  - Si x entró en $\mathtt{S}$ después que w, entonces el algoritmo seteó $\pi[u] = \pi[w] + l(w, u)$ en la iteración en la que se agregó a w. Luego, en la iteración en la que agregó a x concluyó que

    $$\pi[u] = \pi[w] + l(w, u) \leq \pi[x] + l(x, u)$$

    Osea que P es un camino mas corto que P'.
  - Si x entró antes que w, entonces el algoritmo seteó $\pi[u] = \pi[x] + l(x, u)$ en la iteración en la que se agregó a x. Luego, en la iteración en la que agregó a w concluyó que

    $$\pi[u] = \pi[x] + l(x, u) \geq \pi[w] + l(w, u)$$

    Osea que el nuevo camino P es más corto que P' y termina quedando $\pi[u] = \pi[w] + l(w, u)$.

- Si $y \neq u$, entonces ambos vértices son candidatos para ser elegidos en la iteración k. Sin embargo, como se elgió a u vale que:

  $$\pi[w] + l(w, u) = \pi_{k-1}[u] \leq \pi_{k-1}[y]$$

  Sea x el predecesor de y, como $x \in S_{k-1}$ sabemos que $\pi_{k-1}[y] \leq \pi[x] + l(x, y)$ (Hizo esta comparación cuando se agregó x a $\mathtt{S}_{k-1}$).

Ahora por hipotesis inductiva, $\pi[x]$ es la longitud del camino mínimo desde v hasta x entonces:

$$\begin{aligned} l(P) &= \pi[w] + l(w,u) = \pi_{k-1}[u] \leq \pi_{k-1}[y] \leq \pi[x] + l(x,u) \\ &\leq l(P'_{vx}) \leq l(P'_{vx}) + l(x,y) \leq l(P'_{vy}) \leq l(P') \end{aligned}$$

Entonces $l(P) \leq l(P')$ para todo camino P' desde v a u y $\pi[u]$ es la longitud de camino mínimo desde v a u. ■

Teorema 9. *Dado un digrafo $G = (V, X)$ con pesos positivos en los arcos y $v \in V$, el algorimo de Dijkstra determina el camino mínimo entre el vértice v y el resto de los vértices.*

DEMOSTRACIÓN

En cada iteración un nuevo vértice es incorporado a `S` y el algortimo termina cuando `S` $= V$.
Por el lema **??**, al finalizar la última iteración del ciclo, el algoritmo de Dijkstra determina, siguiendo hacia atrás `pred` hasta llegar a v, un camino mínimo entre el vértice v y cada vértice $u \in V$ con longitud $\pi[u]$.
■

Complejidad del algoritmo: Los pasos computacionales críticos son:

1. Encontrar el próximo vértice a agregat en `S`.
2. Actualizar π.

Cada uno de estos pasos se realiza n veces, veamos distintas implementaciones:

- La forma más fácil de implementar **??** es buscar secuencialmente el vértice que minimiza, esto se hace en $O(n)$. En el paso 2, el vértice elegido tiene a lo sumon n vértices adyacentes y para cada uno podemos actualizar en $O(1)$. Considerando esto, cada iteración es $O(n)$, resultando $O(n^2)$ elalgoritmo completo.
- Para mejorar, esta complejidad, en el paso , podemos usar una cola de prioridad implementada sobre un heap de n elementos. Crear esta cola es $O(n)$. Luego es posible borrar el elemento mínimo e insterar uno nuevo en $O(\log n)$. Además, si se mantiene un arreglo auxiliar apuntando a la posición acutal cada vértice en el heap, tambien es posible modificar el valor de π en $O(1)$ y hacer las modificaciones necesarias al heap en $O(logn)$.

  Considerando todas las iteraciones:

  - la operación **??** en $O(logn)$.
  - y $d(u)$ incersiones/modificaciones en el heap para hacer actualizar π en $O(\log n)$ para cada vértice elegido u. Osea que **??** se hace en $O(m * \log n)$.

Luego el algoritmo completo resulta $O(n \log n + m \log n) = O(m \log(n))$. Notar que esta complejidad hace que el algoritmo sea más rápido si $m \in O(n)$ (el grafo es ralo), si $m \in O(n^2)$ (el grafo es denso), entonces la complejidad empeora.

4.1.3. Algoritmo de Bellman-Ford

El algorimo de Bellman-Ford, utiliza programación dinámica para calcular todos los caminos mínimos con origen en un nodo v. Pero, a diferencia de Dijsktra, acepta grafos cuyos arcos pueden tener pesos negativos.

En cada iteración, el algoritmo va guardando una cota superior de la longitud de un camino mínimo desde v a u. Va modificando todas esas cotas de manera iterativa hasta que llega a una iteración en la que ninguna de ellas cambia:

procedure DIJSKTRA($G = (V, X),\ v \in V$)
  $\pi[\mathtt{u}]$ = valor del camino mínimo desde v a u
  $\mathtt{S} \leftarrow \{\mathtt{v}\}$
  $\pi[\mathtt{v}] \leftarrow 0$ ▷ Inicializamos π y *textttpred*
  $\mathtt{pred}[\mathtt{v}] \leftarrow 0$
  for $u \in V$ **do**
    if $(\mathtt{v}, \mathtt{u}) \in \mathtt{X}$ **then**
      $\pi[\mathtt{u}] = \mathtt{l}(\mathtt{v}, \mathtt{u})$
      $\mathtt{pred}[\mathtt{u}] = \mathtt{v}$
    else
      $\pi[\mathtt{u}] = \infty$
      $\mathtt{pred}[\mathtt{u}] = \infty$
    end if
  end for
  while $\mathtt{S} \neq \mathtt{V}$ **do** ▷ Calculamos los caminos mínimos
    $\mathtt{w} \leftarrow \mathtt{arg}\,\mathtt{mín}\{\pi[\mathtt{u}], \mathtt{u} \in \mathtt{V} \backslash \mathtt{S}\}$
    $\mathtt{S} \leftarrow \mathtt{S} \cup \{\mathtt{w}\}$
    for $\mathtt{u} \in \mathtt{V} \backslash \mathtt{S} \ \wedge\ (\mathtt{w}, \mathtt{u}) \in \mathtt{X}$ **do**
      if $\pi[\mathtt{u}] > \pi[\mathtt{w}] + \mathtt{l}(\mathtt{w}, \mathtt{u})$ **then**
        $\pi[\mathtt{u}] \leftarrow \pi[\mathtt{w}] + \mathtt{l}(\mathtt{w}, \mathtt{u})$
        $\mathtt{pred}[\mathtt{u}] = \mathtt{w}$
      end if
    end for
  end while
  return $[\pi, pred]$
end procedure

A. Hoja de complejidades

A.1. Algoritmos sobre arrays

Algoritmo	Complejidad
De Búsqueda	
Secuencial	$O(n)$
Binaria	$O(\log n)$
De Ordenamiento	
Bubblesort	$O(n^2)$
Quicksort	$O(n^2)$
Heapsort	$O(n \log n)$*

* $O(n \log n)$ es la complejidad óptima para algoritmos de ordenamiento basados en comparaciones.

A.2. Algoritmos sobre grafos

	Complejidad		
Algoritmo	**Matriz de adyacencia**	**Lista de Adyacencia**	**Lista de Incidencias**
Árbol generador Mínimo			
Prim	$O(n^2)$	$(n+m)\log m$	$m \log m$
Kruskal	$O(n^2)$	$O(m * log(n))$	$n + m \log(n)$
Camino Mínimo			
BFS	$O(n^2)$	$O(n+m)$	
Dijsktra	$O(n^2)$	$O(n+m)\log n$	

Grafos Definiciones básicas: isomorfismos. Enumeración. Grafos eulerianos y hamiltonianos. Planaridad. Coloreo. Número cromático. Matching, conjunto independiente, recubrimiento. Recubrimiento de aristas y vértices.

Algoritmos en grafos y aplicaciones Representación de un grafo en la computadora: matrices de incidencia y adyacencia, listas. Algoritmos de búsqueda en grafos: A*. Arboles ordenados: códigos unívocamente descifrables. Algoritmos para detección de circuitos. Algoritmos para encontrar el camino mínimo en un grafo: Dijkstra, Ford, Dantzig. Planificación de procesos: PERT/CPM. Algoritmos heurísticos: ejemplos. Nociones de evaluación de heurísticas y de técnicas metaheurísticas. Algoritmos aproximados. Heurísticas para el problema del viajante de comercio. Algoritmos para detectar planaridad. Algoritmos para coloreo de grafos. Algoritmos para encontrar el flujo máximo en una red: Ford y Fulkerson. Matching: algoritmos para correspondencias máximas en grafos bipartitos. Otras aplicaciones.

Problemas NP-completos Problemas tratables e intratables. Problemas de decisión. P y NP. Maquinas de Turing no determinísticas. Problemas NP-completos. Relación entre P y NP. Problemas de grafos NP-completos: coloreo de grafos, grafos hamiltonianos, recubrimiento mínimo de las aristas, corte máximo, etc.