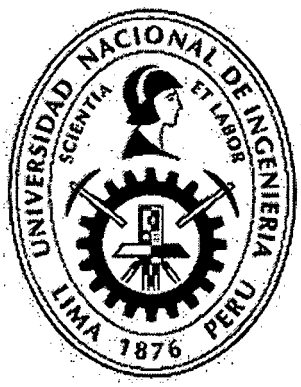
UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Matemática



Tesis para optar el título profesional de:

Licenciado en Matemática

Titulada:

COMPARACION DE LOS ALGORITMOS PRIM Y KRUSKAL

Presentada por:

Paul Miller Tocto Inga

Asesor:

Mg. William C. Echegaray Castillo

LIMA-PERÚ

Octubre, 2012

Dedicado a
mi familia: mi querida esposa Luz, por su amor y apoyo constante
y a mis hijos Omar, Samir, Felix y Paul, por la alegría que me
brindan cada día

Agradecimientos

Muchas gracias a todos los que me ayudaron para que sea posible este trabajo de investigación, en especial al Profesor William Echegaray.

Resumen

Como los algoritmos son la base de la ciencia de la computación y una de sus disciplinas es la complejidad de los algoritmos, rama que necesita una base de análisis matemático, lo cual es tratado en la siguiente investigación, con el objetivo en primer término, de describir los fundamentos de los algoritmos, describiendo las técnicas existentes de diseño para crear algoritmos, a continuación las características de un buen algoritmo y las aplicaciones de los algoritmos, y también se describe las técnicas matemáticas para realizar el análisis de algoritmos, basándose en la definición de las funciones de complejidad asociados al algoritmo que se está analizando, a continuación se detalla los fundamentos de la teoría de grafos necesarios para el estudio de los algorítmos PRIM y KRUSKAL, definidos mediante la teoría de grafos, concluyendo con la comparación de la complejidad de los mismos.

Índice general

Índice de figuras

Capítulo 1

Fundamentos de Algoritmos

Los algoritmos son fundamentales en todas las ramas de la ciencia, en particular creo que es importante conocer los fundamentos de la teoría de algoritmos para aplicarlos en la solución de problemas reales, empezaremos con los fundamentos básicos, luego se realizará una introducción al diseño de algoritmos.

1.1. Algoritmos

La palabra algoritmo es de origen árabe, proviene de la palabra al-Khwarizmi, sobrenombre del famoso matemático Mohamed Ben Musa, que quiere decir de Khwarizmi, el estado donde nació Ben Musa.

Mohamed Ben Musa fue un matemático famoso, quien estableció los métodos básicos para sumar, multiplicar y dividir números, raíz cuadrada y el cálculo de los dígitos de pi. Estos procedimientos eran precisos, claros, mecánicos y correctos, que definitivamente eran algoritmos, razón por lo cual el nombre se puso en honor a él.

Como regla mnemotécnica, se dice que un algoritmo es un fideo: finito, definido y organizado.

1.1.1. Definición de Algoritmo

El Diccionario de la Lengua Española de la Real Academia Española define el término *algoritmo* como un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema. Entonces se define en matemáticas un *algoritmo* como cualquier procedimiento formado por un conjunto finito de instrucciones no ambiguas y efectivas bien definidas que toma de cero a más valores como entrada y genera algún valor, o un conjunto de valores como salida, entonces un algoritmo es una secuencia de pasos que transforma las entradas en salidas y para ejecutarse necesitan una cantidad finita de recursos:

- Tiempo
- Memoria

Otra característica que se deduce de la definición es que un mismo algoritmo puede resolver todos los problemas de una misma clase.

Se debe de considerar en identificar los problemas que resolverá el algoritmo. Aquellos problemas bien definidos y adecuados para ser solucionados mediante el uso de computadoras digitales.

Los algoritmos se escriben pensando en el ejecutor o procesador del algoritmo(máquina o personal). El procesador debe de ser capaz de interpretarlo y de ejecutarlo.

1.1.2. Aplicaciones de los algoritmos

Los algoritmos están presente en todos los campos, así tenemos[1]:

- El proyecto del genoma humano tiene por objetivo identificar todos los 100,000 genes, en el ADN humano, determinar la secuencia de los 3 billones de pares de químicos base que forman parte el ADN humano, guardar la

información en la base de datos y el desarrollo de herramientas para el análisis de la data, en cada una de estas etapas es necesario algoritmos sofisticados y complejos. Los ahorros por el uso de los algoritmos en la solución de problemas biológicos son: tiempo de horas hombres, tiempo de horas máquina y dinero.

- Internet permite a todas las personas alrededor del mundo acceder rápidamente y recuperar cantidades grandes de información, para lo cual se necesita algoritmos que permita gestionar y manipular los grandes volúmenes de información. Ejemplos de problemas que deben de ser resueltos son: hallar rutas adecuadas para la transferencia de los datos y el uso de buscadores para hallar rápidamente las páginas en las cuales se encuentra una determinada información.

- El comercio electrónico permite negociar e intercambiar bienes y servicios electrónicamente. Mantener de forma confidencial el número de la tarjeta de crédito, los passwords y las reglas de negocio del banco es fundamental para el uso masivo del comercio electrónico, usando para tal efecto llaves públicas criptográficas y firmas digitales, los cuales están basadas en algoritmos numéricos y la teoría de números.

- En la fabricación de bienes es importante asignar los recursos escasos disponibles en la forma más beneficiosa posible, usando para tal fín la programación lineal, algunos ejemplos son:

    - Una compañía de petróleo necesita saber donde ubicar su pozos para maximizar sus ganancias.
    - Un candidato presidencial deseará determinar donde invertir en publicidad, con la finalidad de maximizar su posibilidad de ganar las elecciones.

- Un proveedor de servicios de internet desearía determinar, donde ubicar recursos adicionales con el objetivo de servir a sus clientes con mayor efectividad.
- Una aereolía desearía asignar las tripulaciones de vuelos con el menor costo posible, asegurándose que cada vuelo esté cubierto y que las regulaciones del gobierno con respecto a las tripulaciones se cumplan.

1.2. Diseño de los algoritmos

Existen muchas técnicas para diseñar algoritmos, algunas de las cuales son las siguientes: Divide y vencerás, programación dinámica, ávidos y método de retroceso (backtraking, poda alfa).

1.2.1. Divide y vencerás

Cuando el algoritmo es recursivo en estructura, para resolver un determinado problema, se llaman a sí mismo recursivamente uno o mas veces, esta técnica que consiste en la descomposición de un problema de tamaño n en problemas mas pequeños independientes, de tal forma que usando la solución independiente de dichos problemas sea posible construir con facilidad una solución al problema completo, también es importante siempre balancear los costos, siempre que sea posible, por lo tanto es mejor que los subproblemas generados con la aplicación de esta técnica siempre tengan un tamaño aproximadamente igual.

Ejemplo de problemas que se resuelven con está técnica tenemos a: multiplicación de enteros grandes, programación de torneos de tenis, búsqueda binaria, ordenación por mezcla, ordenación rápida, búsqueda de la mediana, multiplicación de matrices, etc.

La resolución de un problema mediante esta técnica consta de los siguientes pasos:

1. En primer lugar se debe de plantear el problema de tal forma que pueda ser descompuesta en k subproblemas del mismo tipo, pero de menor tamaño. Es decir, si el tamaño de la estrada es n. Se debe de conseguir dividir el problema en k subproblemas (donde $1 \leq k \leq n$), cada uno con una entrada de tamaño n_k y donde $0 \leq n_k < n$. A esta tarea se le conoce como división.

2. En segundo lugar han de resolverse independientemente todos los subproblemas, bien directamente si son elementales o bien de forma recursiva. El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema nos garantiza la convergencia hacia los casos elementales, también denominados casos base.

3. Combinar las soluciones hallados a los sub problemas para la solución del problema original.

Algunas consideraciones a tomar en cuenta al usar esta técnica son las siguientes:

1. El número k debe de ser pequeño e independiente de una entrada determinada. El caso particular cuando el algoritmo tiene una sola llamada recursiva, es decir $k = 1$, le denominaremos algoritmo de simplificación, ejemplos de este tipo de algoritmos son: el cálculo del factorial de un número, búsqueda binaria en un vector, etc.

2. La ventaja de los algoritmos de simplificación es que consiguen reducir el tamaño del problema en cada paso, por lo que sus tiempos de ejecución suelen ser muy buenos (normalmente de orden logaritmico o lineal). Además

pueden admitir una mejora adicional, por cuanto se puede eliminar fácilmente la recursión mediante el uso de un bucle iterativo, lo que dá como resultado menores tiempos de ejecución y menor complejidad espacial al no utilizar la pila de recursión, pero esto ocasiona un detrimiento de la legibilidad del código resultante.

3. El diseño usando esta técnica es simple, claro, robusto y elegante, lo cual redunda en una mayor legibilidad y facilidad de depuración y mantenimiento del código obtenido.

4. La desventaja es que los diseños recursivos conllevan normalmente a un mayor tiempo de ejecución que los iterativos, además de la complejidad espacial que puede representar el uso de la pila de recursión.

5. Para obtener eficiencia, es importante conseguir que los subproblemas sean independientes o que no exista solapamiento entre ellos. De lo contrario el tiempo de ejecución de estos algortimos será exponencial.

6. El número de subproblemas y su tamaño, influye en la complejidad del algoritmo resultante, afectando la eficiencia.

7. Es importante que la división en subproblemas se realice de la forma mas equilibrada posible, para evitar anomalías de funcionamiento, lo cual significa el aumento de la complejidad.

1.2.2. Programación dinámica

Este método particiona el problema en subproblemas dependientes, resuelve los subproblemas recursivamente y entonces combina sus soluciones para resolver el problema original, los subproblemas dependientes comparten sus subsubproblemas, los cuales son resueltos una sola vez y son guardados para los siguientes

cálculos en una tabla de soluciones, evitando el trabajo de recalcular cada vez que el subsubproblema es encontrado. La formación de la tabla de subproblemas para alcanzar una solución a un problema dado se denomina programación dinámica, nombre que proviene de la teoría de control [1].

La forma de un algoritmo de programación dinámica puede variar, pero es común lo siguiente: una tabla a rellenar y un orden en el cual se hacen las entradas.

Este método se aplica a problemas de optimización.

El desarrollo de un algoritmo de programación dinámica puede ser divida en una secuencia de cuatro etapas:

- Caracterizar la estructura de una solución optimal.
- Recursivamente definir el valor de una solución óptima.
- Calcular el valor de una solución óptima en el sentido de abajo hacia arriba.
- Construir una solución óptima de la información calculada

Se pueden aplicar en la solución de los siguientes problemas: Problema de triangulación, apuestas en la serie mundial de béisbol, multiplicación de matrices, etc.

1.2.3. Avidos o Voraces (greedy)

Esta técnica selecciona la opción que sea localmente óptimal en cualquier sentido particular y en cualquier paso individual que se dé, ignorando el efecto futuro. Es adecuado el uso de esta técnica cuando podamos garantizar que la mejor solución local nos lleve a la solución global del problema, además de servir para obtener rápidamente una aproximación a la solución del problema, tener presente que la solución no siempre será la optima, por ejemplo al considerar

aristas ponderadas negativas en los algoritmos de Dijkstra y de Kruskal, tenemos como resultado que el algoritmo de Kruskal no se verá afectado mostrando siempre la solución correcta, pero en cambio el algoritmo de Dijkstra en algunos casos no produce la solución correcta. Se pueden aplicar en problemas de optimización: búsqueda del camino más corto, búsqueda del árbol de menor o mayor peso.

Descripción del Algoritmo.-

1. Para resolver el problema, un algoritmo ávido tratará de encontrar un subconjunto de candidatos tales que, cumpliendo las restricciones del problema, constituya la solución óptima.

2. Para ello trabajará por etapas, tomando en cada una de ellas la decisión que le parece la mejor, sin considerar las consecuencias futuras, y por tanto escogerá de entre todos los candidatos el que produce un óptimo local para esa etapa, suponiendo que será a su vez óptimo global para el problema.

3. Antes de añadir un candidato a la solución que está construyendo comprobará si es prometedora al añadirlo. En caso afirmativo lo incluirá en ella y en caso contrario descartará este candidato para siempre y no volverá a considerarlo.

4. Cada vez que se incluye un candidato comprobará si el conjunto obtenido es solución.

Para identificar si un problema es factible de ser resuelto por un algoritmo ávido, deben estar definidos los siguientes componentes, los cuales tienen que estar presente en el problema.

1. Un conjunto de candidatos.- Es el conjunto de donde se crea la solución.

2. Una función de selección.- Escoge el mejor candidato a ser adicionado a la solución, entre los que aún no han sido seleccionados ni rechazados.

3. Una función de factibilidad.- Verifica si un cierto subconjunto de candidatos se usa para determinar si un candidato puede formar parte de la solución.

4. Una función objetivo.- Asigna un valor a la solución o a la solución parcial hallada. Es la función que queremos maximizar o minimizar.

5. Una función solución.- Comprueba si un subconjunto de las entradas es solución al problema , sea óptima o no.

Consideraciones

1. No se puede utilizar siempre esta técnica porque no todos los problemas admiten esta estrategia de solución.

2. Se debe de probar que la función de selección nos permita encontrar óptimos globales, para cualquier entrada del algoritmo.

3. También se utiliza para poder hallar una solución aproximada al problema, que servirá luego para poder hallar la solución definitiva usando otras técnicas.

4. Su nombre original proviene del término ingles *greedy*, que siginifica que el algoritmo en cada etapa "toman lo que pueden"sin analizar consecuencias, es decir son glotones por naturaleza.

1.3. Características de los algoritmos

Para diseñar un algoritmo correctamente se debe buscar que se cumpla las siguientes características:

- Existencia de un conjunto de entradas.- Debe de existir un conjunto específico de objetos cada uno de los cuales son los datos iniciales de un caso particular del problema que resuelve el algoritmo. Este conjunto se llama conjunto de entradas del problema.

- Finitud.- Un algoritmo siempre tiene que finalizar tras un número finito de pasos. Esto significa que el tiempo de ejecución de un algoritmo es finito.

- Precisión.- Todas las acciones de un algoritmo deben estar bien definidas, no deben ser ambiguas, cada una de ellas solo debe de interpretarse de una forma única .

- Predecibilidad.- Dado un conjunto de entradas y si ejecutamos el algoritmo muchas veces con el mismo conjunto de entradas, obtendremos siempre el mismo resultado intermedio y final.

- Claridad.- Un algoritmo tiene que ser comprensible para el ser humano, para poder implementarlo posteriormente con un lenguaje de programación.

- Generalidad.- Todos los algoritmos tienen que resolver problemas generales, es decir los valores de las entradas tienen que ser variables, que pueden tomar los valores correspondientes a su domino.

- Eficiencia.- Se debe buscar que el algoritmo al momento de ser implementado con un software, use los recursos computacionales mínimos: Memoria, CPU, Tiempo.

- Sencillez.- El algoritmo tiene que ser lo mas simple posible o lo menos complejo, se debe de buscar un equilibrio entre la claridad y la eficiencia, aun si se pierde un poco de eficiencia.

- Modularidad.- Considerar que el algoritmo puede formar parte de un Algoritmo mucho más complejo, como también que el algoritmo puede ser descompuesto en otros algoritmos, siempre y cuando esta descomposición favorezca a la claridad del algoritmo.

- Efectividad.- Todas las operaciones a realizar en el algoritmo deben ser suficientemente básicas para que se pueda hacer en un tiempo finito en el procesador que ejecuta el algoritmo.

- Existencia de un conjunto de salidas.- Es el conjunto producto del resultado del algoritmo asociado al conjunto de entradas.

Capítulo 2

Análisis de la complejidad de algoritmos

En este capítulo se desarrolla la teoría de la complejidad en la parte que corresponde al análisis del tiempo de ejecución o complejidad temporal.

2.1. Análisis de algoritmos

Analizar un algoritmo es predecir los recursos que el algoritmo requiere, recursos como memoria, ancho de banda de comunicaciones o hardware necesario, pero frecuentemente lo que se desea medir es el tiempo computacional, para este fin en la ciencia computacional existe una rama llamada Análisis de la complejidad de algoritmos, que forma parte de la teoría de la complejidad computacional, que tiene como objetivo la estimación teórica de los recursos necesarios para que un algoritmo resuelva un problema computacional dado, buscando con este fin la eficiencia del algoritmo.

Existen dos parámetros que son usados frecuentemente con los cuales podemos buscar la eficiencia: el tiempo de ejecución (complejidad temporal) y la

cantidad de memoria (espacio, complejidad espacial), en el presente estudio se toma en cuenta el parámetro tiempo de ejecución, que es el más usado.

El tiempo de ejecución de un algoritmo dependerá de diversos factores tales como: los datos de entrada, la calidad del código generado por el compilador para crear el programa objeto, la naturaleza y rapidez de las instrucciones máquina del procesador concreto que ejecute el programa, y la complejidad intrínseca del algoritmo. Existen dos estudios posibles sobre el tiempo [3]:

- Uno que proporciona una medida teórica (a priori), que consiste en obtener una función que acote (por arriba o por abajo) el tiempo de ejecución del algoritmo para unos valores de entrada dados.
- Y otro que ofrece una medida real (a posteriori), que consiste en medir el tiempo de ejecución del algoritmo para unos valores de entrada dados y en un ordenador concreto, por ejemplo: PENTIUM DUAL CORE, AMD, etc.

Para analizar la complejidad usando el tiempo, dependiendo del problema se considera una variable que mida su tamaño:n, esta variable dependerá de la naturaleza del problema, que es el número de componentes sobre los que se va a ejecutar el algoritmo, así tenemos los siguientes ejemplos:

- Para un vector será su longitud
- Para una matriz será el número de elementos
- Para un grafo se considera el número de vértices o el número de arcos
- En un archivo de datos se toma en cuenta el número de registros

Una vez definida la variable n, se busca una función al cual llamaremos función de complejidad temporal que depende de la variable n, que tiene como dominio los números naturales $N = \{0, 1, 2, \} : f(n)$, que mida la complejidad

temporal del algoritmo, para que sea independiente del tipo de máquina, lenguaje de programación, compilador, etc. en el cual se implementará el algoritmo, esta función teóricamente indica el número de instrucciones ejecutadas por un ordenador idealizado.

El análisis se realizará en el comportamiento de la función de complejidad temporal: $f(n)$, para grandes valores de n.

2.2. Mejor caso vs peor caso

Se realiza un análisis del peor caso, caso medio y del mejor caso, para analizar la eficiencia, frecuentemente se analiza el peor caso, también se tiene que tener presente que muchas veces un algoritmo más ineficiente puede ser el más adecuado para resolver un problema real, porque existen otros factores que se deben de considerar.

2.2.1. Peor caso

Es el análisis más frecuente realizado porque en ningún caso el costo de cualquier caso excederá este valor, porque es el caso correspondiente al de mayores pasos, consiste en el análisis del tiempo máximo necesario para un problema de tamaño n, con las siguientes consideraciones:

- Analizar el crecimiento de la función de complejidad cuando el tamaño aumenta.
- Ignorar las constantes y los términos de menor peso dependientes del contexto.

También el tiempo del peor caso en algunos algoritmos ocurre frecuentemente, por ejemplo en el la búsqueda de información en una base de datos, el peor caso

ocurre cuando la información no se encuentra en la base de datos.

2.2.2. Caso medio

Es el análisis que consiste en el análisis del tiempo medio esperado para un problema cualquiera de tamaño n, este caso muchas veces es similar al peor caso.

2.2.3. Mejor caso

Este es un análisis que no se realiza frecuentemente porque puede ser engañoso y es un caso ideal, y que en la práctica como pueden ocurrir todos los casos, es mejor estudiar el del peor caso.

2.3. Tamaño de la entrada y tiempo computacional

El tamaño dependerá del problema en estudio, así tenemos:

PROBLEMA	TAMAÑO DEL PROBLEMA
Búsqueda de un elemento en un conjunto	Número de elementos en el conjunto
Multiplicar dos matrices	Dimensión de las matrices
Recorrer un árbol	Número de nodos en el árbol
Resolver un sistema de ecuaciones lineales	Número de ecuaciones y/o incógnitas
Ordenar un conjunto de valores	Número de elementos en el conjunto
Multiplicación de dos números	Número de bits de los números

Entonces debemos ser explícitos en los datos de la entrada, dependiendo del problema. El tiempo computacional es el número de pasos ejecutados o el número de operaciones, de esta manera tendremos independencia de las computadoras,

para realizar un mejor análisis se trabaja con el concepto de operaciones elementales.

2.3.1. Operaciones elementales

Es la operación que realiza un computador en un tiempo acotado por una constante, por ejemplo tenemos:

- Operaciones aritméticas básicas
- Asignaciones a variables de tipo predefinido por el compilador
- Llamadas a funciones y procedimientos
- Retorno de los procedimientos
- Comparaciones lógicas
- Acceso a estructuras indexadas, tales como los vectores y matrices

2.3.2. Tiempo de ejecución

El tiempo de ejecución que tiene un algoritmo, se puede modelar mediante una función que mida el número de operaciones elementales que realiza un algoritmo para un tamaño de entrada dado, al cual se le denomina función de complejidad temporal, que permitirá el estudio de la complejidad del algoritmo.

2.4. Funciones de complejidad

La función de complejidad temporal mide el orden del crecimiento del tiempo de ejecución de un algoritmo, lo cual nos permitirá analizar la eficiencia del algoritmo y también nos permitirá comparar el desempeño de algoritmos alternativos, para poder implementar el más óptimo.

Aún cuando se pueda algunas veces determinar el tiempo exacto de ejecución de un algoritmo, mediante una función de complejidad temporal totalmente definida, por otro lado cuando las entradas son grandes, las constantes multiplicativas que existen en la función de complejidad temporal y todos los términos de orden menor al mayor que exista en la función, son dominados por los efectos del tamaño de la entrada.

El estudio asintótico de los algoritmos consiste en el análisis del tamaño de las entradas suficientemente grandes para hacer que el orden de crecimiento del tiempo de ejecución sea lo más relevante, analizando para tal efecto como el tiempo de ejecución de un algoritmo se incrementa con el tamaño de la entrada en el límite, sin ninguna restricción, usualmente los algoritmos que son asintoticamente mas eficiente será la mejor elección pero para muy pocas entradas.

2.4.1. Notación asintótica

Las funciones de complejidad temporal asociadas a un algoritmo se estudiarán mediante otras funciones de referencia, que permiten comparar la rapidez del crecimiento de un par de funciones, para lo cual se define las siguientes notaciones siguientes:

notación Θ

Esta es una acotación por arriba y debajo de una función, se define de la siguiente forma:

$\Theta(g(n)) = \{f(n)/Existen\ constantes\ positivas\ c_1, c_2\ y\ n_0 \in \mathbb{N}\ tal\ que\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0\}$

Se considera equivalentemente lo mismo a las siguientes expresiones:

- $f(n) \in \Theta(g(n))$

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

Donde $f(x) = \Theta(g(x))$ se puede interpretar que f crece a la misma razón que g cuando x es muy grande [5].

Gráficamente sería de la siguiente forma:

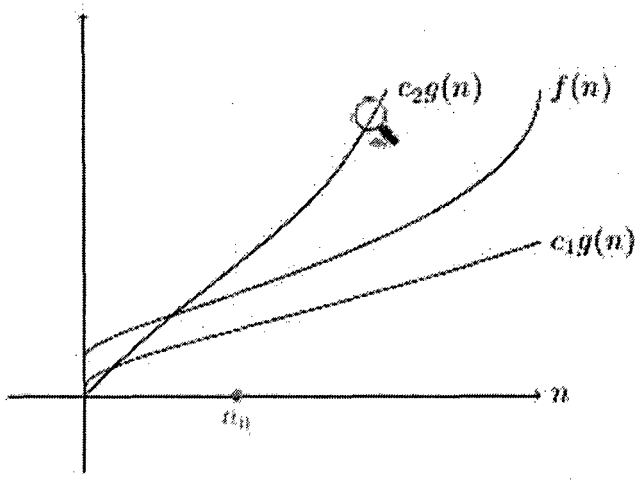


Figura 2.1: Notación asintótica

Propiedades de Θ

1. Para cualquier función f se tiene que $f \in \Theta(f)$

   Demostración:

   Para cualquier función f se cumple: $f(n) \leq f(n) \leq f(n)\ \forall n$

   entonces existen $c_1 = c_2 = 1$ tal que $c_1 f(n) \leq f(n) \leq c_2 f(n)\ \forall n$

   entonces $f(n) \in \Theta(g(n))$

2. $f \in \Theta(g) \Rightarrow \Theta(f) = \Theta(g)$

   Demostración:

a) ($\Rightarrow$) $\Theta(f) \subset \Theta(g)$

Por (1) se cumple $f \in \Theta(f)$

y como dato $f \in \Theta(g)$

entonces se cumple: $\Theta(f) \subset \Theta(g)$

b) ($\Leftarrow$) $\Theta(f) \supset \Theta(g)$

Por (1) se cumple $g \in \Theta(g)$

Por dato se cumple $f \in \Theta(g)$, por lo tanto

existen c_1, c_2, n_0 tal que $c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0$

$\Rightarrow \frac{1}{c_2} f(n) \leq g(n) \leq \frac{1}{c_1} f(n)\ \forall n \geq n_0$

entonces por definición se cumple $g(n) \in \Theta(f(n))$

De a) y b) Se concluye que $\Theta(f) = \Theta(g)$

3. $\Theta(f) = \Theta(g) \Leftrightarrow f \in \Theta(g)$ y $g \in \Theta(f)$

4. $f \in \Theta(g)$ y $g \in \Theta(h) \Rightarrow f \in \Theta(h)$

   Si $f \in \Theta(g)$ se cumple:

   existen c_1, c_2, n_0 tal que $c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0$ (1)

   Si $g \in \Theta(h)$ se cumple:

   existen d_1, d_2, m_0 tal que $d_1 h(n) \leq g(n) \leq d_2 h(n)\ \forall n \geq m_0$ (2)

   de (1) y (2) obtenemos:

   $c_1 d_1 h(n) \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \leq c_2 d_2 h(n)\ \forall n \geq max\{n_0, m_0\}$

   entonces existen $e_1 = c_1 d_1, e_2 = c_2 d_2, l_0 = max\{n_0, m_0\}$ y se cumple:

   $e_1 h(n) \leq f(n) \leq e_2 h(n)\ \forall n \geq l_0$

   $\Rightarrow f(n) \in \Theta(h)$

5. Regla de la suma: Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h) \Rightarrow f_1 + f_2 \in \Theta(max(g, h))$

6. Regla del producto: Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h) \Rightarrow f_1 f_2 \in \Theta(g.h)$

   Si $f_1 \in \Theta(g)$ se cumple:

   existen c_1, c_2, n_0 tal que $c_1 g(n) \leq f_1(n) \leq c_2 g(n)\ \forall n \geq n_0$ (1)

   Si $f_2 \in \Theta(h)$ se cumple:

   existen d_1, d_2, m_0 tal que $d_1 h(n) \leq f_2(n) \leq d_2 h(n)\ \forall n \geq m_0$ (2)

   de (1) y (2) obtenemos:

   $c_1 d_1 g(n)h(n) \leq f_1(n)f_2(n) \leq c_2 d_2 g(n)h(n)\ \forall n \geq max\{n_0, m_0\}$

   entonces existen $e_1 = c_1 d_1, e_2 = c_2 d_2, l_0 = max\{n_0, m_0\}$ y se cumple:

   $e_1 g(n)h(n) \leq f_1(n)f_2(n) \leq e_2 g(n)h(n)\ \forall n \geq l_0$

   $\Rightarrow f_1(n)f_2(n) \in \Theta(gh)$

7. Si existe $lim_{n\to\infty} \dfrac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:

   - *Si* $k \neq 0$ y $k < \infty \Rightarrow \Theta(f) = \Theta(g)$
   - *Si* $k = 0$ las órdenes exactos de f y g son distintos.

notación $\mathcal{O}$

Dada una función f, queremos estudiar aquellas funciones g que a lo sumo crecen tan deprisa como f. Al conjunto de tales funciones se le llama cota superior de f y lo denominamos $\mathcal{O}(f)$. Conociendo la cota superior de un algoritmo podemos asegurar que, en ningún caso, el tiempo empleado será de un orden superior al de la cota.

También se puede considerla como una acotación por arriba solamente de una función y se define de la siguiente forma:

Sea $f, g : Z^+ \to R$, *Se dice que g domina a f o f es dominada por g si existen*

constantes $m \in R^+$ *y* $k \in Z^+$ *tal que* $|f(n)| \leq m|g(n)|\ \forall n \in Z^+,\ n \geq k$

También algunos autores consideran:

f es dominado por g $\iff$ *f es de orden a lo mas de g* $\iff$ $f \in \mathcal{O}$

También llamado: Big-$\mathcal{O}$

Donde:

$\mathcal{O}(g)$: *Orden g, Big*$-\mathcal{O}$ *de g, representa al conjunto de funciones dominadas por g.*

Se considera equivalentemente lo mismo a las siguientes expresiones:

- $f(n) \in \mathcal{O}(g(n))$
- $f(n) = \mathcal{O}(g(n))$

Donde $f(x) = \mathcal{O}(g(x))$ se puede interpretar que f crece a la misma razón o que crece mas despacio que g cuando x es muy grande [5].

Gráficamente sería de la siguiente forma:

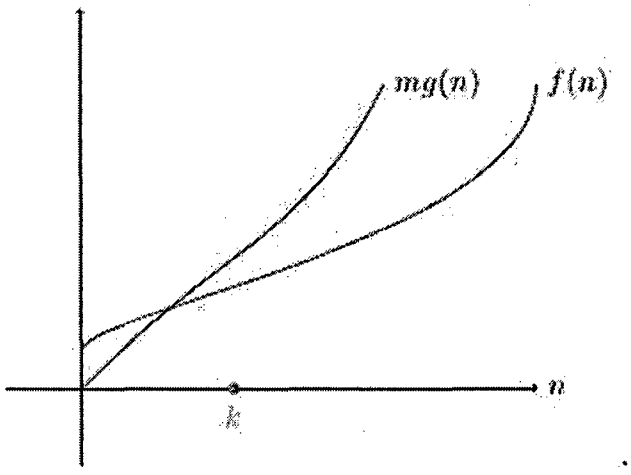


Figura 2.2: $\mathcal{O}$ notación

Esta es una de las más utilizadas en el estudio de la complejidad de un algoritmo.

Algunas funciones especiales Big-$\mathcal{O}$:

BIG-$\mathcal{O}$	NOMBRE
$\mathcal{O}(1)$	Constante
$\mathcal{O}(log_2 n)$	Logarítmico
$\mathcal{O}(n)$	Lineal
$\mathcal{O}(nlog_2 n)$	$O(nlog_2 n)$
$\mathcal{O}(n^2)$	Cuadrático
$\mathcal{O}(n^3)$	Cúbico
$\mathcal{O}(n^m),\ m = 0, 1, 2, 3, ...$	Polinomial
$\mathcal{O}(c^n),\ c > 1$	Exponenciall
$\mathcal{O}(n!)$	Factorial

El análisis de algunas de las funciones especiales segun Sedgewick[4] es:

- Constante: Es cuando una instrucción se ejecuta una vez o a lo mas solamente unas cuantas veces.
- $logN$: Cuando N se duplica $logN$ se incrementa por una constante y $logN$ se duplica cuanto N se incrementa hasta N^2
- N: Cuando el tiempo de un algoritmo es lineal, es la situación óptima que indica que el algoritmo debe procesar N entradas.
- $NlogN$: Tenemos una combinación de las dos funciones anteriores, cuando N se duplica el tiempo llega a ser mas que el doble.
- N^2: Es cuando se tiene el proceso de todos los pares de items de datos. Cuando N se duplica, el tiempo se llega a cuadruplicar.
- N^3: Es cuando se tiene el proceso triple de los items de datos. Cuando N se duplica, el tiempo llega al valor de ocho veces el valor original.

- 2^N: Este valor indica la solución de los problemas usando la fuerza bruta. Cuando N se duplica, el tiempo llega al valor del cuadrado del valor original.

Propiedades de $\mathcal{O}$

1. Para cualquier función f se tiene que $f \in \mathcal{O}(f)$
2. $f \in \mathcal{O}(g) \Rightarrow \mathcal{O}(f) \subset \mathcal{O}(g)$
3. $\mathcal{O}(f) = \mathcal{O}(g) \Leftrightarrow f \in \mathcal{O}(g)$ y $g \in \mathcal{O}(f)$
4. $f \in \mathcal{O}(g)$ y $g \in \mathcal{O}(h) \Rightarrow f \in \mathcal{O}(h)$
5. $f \in \mathcal{O}(g)$ y $g \in \mathcal{O}(h) \Rightarrow f \in O(min(g,h))$
6. Regla de la suma: Si $f_1 \in \mathcal{O}(g)$ y $f_2 \in \mathcal{O}(h) \Rightarrow f_1 + f_2 \in \mathcal{O}(max(g,h))$
7. Regla del producto: Si $f_1 \in \mathcal{O}(g)$ y $f_2 \in \mathcal{O}(h) \Rightarrow f_1 f_2 \in \mathcal{O}(g.h)$
8. Si existe $lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:
   - $Si\ k \neq 0$ y $k < \infty \Rightarrow \mathcal{O}(f) = \mathcal{O}(g)$
   - $Si\ k = 0 \Rightarrow f \in \mathcal{O}(g)$, es decir. $\mathcal{O}(f) \subset \mathcal{O}(g)$, pero sin embargo se verifica que $g \notin \mathcal{O}(f)$

notación Ω

Dada una función f, queremos estudiar aquellas funciones g que a lo sumo crecen tan lentamente como f. Al conjunto de tales funciones se le llama cota inferior de f y lo denominamos $\Omega(f)$. Conociendo la cota inferior de un algoritmo podemos asegurar que, en ningún caso, el tiempo empleado será de un orden inferior al de la cota.

Esta es una acotación por debajo de una función, se define de la siguiente forma:

$\Omega(g(n)) = \{f(n)/$ *Existen constantes positivos* c *y* n_0 *tal que* $0 \leq cg(n) \leq f(n)\ \forall\ n \geq n_0\}$

Se considera equivalentemente lo mismo a las siguientes expresiones:

- $f(n) \in \Omega(g(n))$
- $f(n) = \Omega(g(n))$

Donde $f(x) = \Omega(g(x))$ se puede interpretar como la negación de la interpretación de o cuando x es muy grande [5].

Gráficamente sería de la siguiente forma:

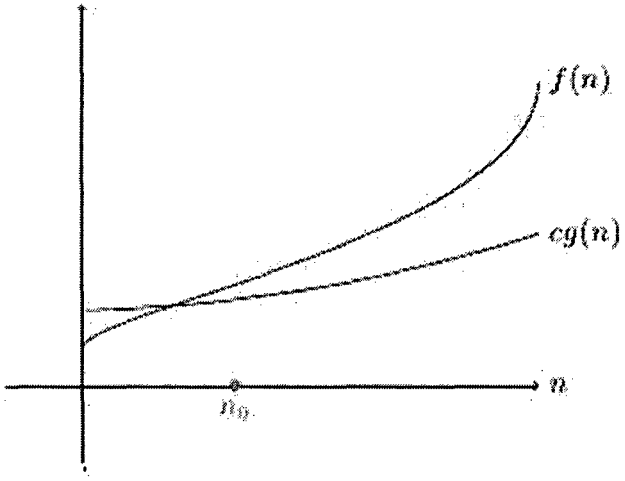


Figura 2.3: Ω notación

Propiedades de Ω

1. Para cualquier función f se tiene que $f \in \Omega(f)$

   Demostración:

Para cualquier función f se cumple: $f(n) \leq f(n)\ \forall n$

entonces existen $c_1 = 1$ tal que $c_1 f(n) \leq f(n)\ \forall n$

entonces $f(n) \in \Omega(f(n))$

2. $f \in \Omega(g) \Rightarrow \Omega(f) \subset \Omega(g)$

   Demostración:

   Por (1) se cumple $f \in \Omega(f)$

   y como dato $f \in \Omega(g)$

   entonces se cumple: $\Omega(f) \subset \Omega(g)$

3. $\Omega(f) = \Omega(g) \Leftrightarrow f \in \Omega(g)$ y $g \in \Omega(f)$

4. $f \in \Omega(g)$ y $g \in \Omega(h) \Rightarrow f \in \Omega(h)$

   Si $f \in \Omega(g)$ se cumple:

   existen c, n_0 tal que $cg(n) \leq f(n)\ \forall n \geq n_0$ (1)

   Si $g \in \Omega(h)$ se cumple:

   existen d, m_0 tal que $dh(n) \leq g(n)\ \forall n \geq m_0$ (2)

   de (1) y (2) obtenemos:

   $cdh(n) \leq cg(n) \leq f(n)\ \forall n \geq max\{n_0, m_0\}$

   entonces existen $e = cd, l_0 = max\{n_0, m_0\}$ y se cumple:

   $eh(n) \leq f(n)\ \forall n \geq l_0$

   $\Rightarrow f(n) \in \Omega(h)$

5. $f \in \Omega(g)$ y $g \in \Omega(h) \Rightarrow f \in \Omega(max(g, h))$

6. Regla de la suma: Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Rightarrow f_1 + f_2 \in \Omega(g + h)$

7. Regla del producto: Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Rightarrow f_1 f_2 \in \Omega(g.h)$

   Si $f_1 \in \Omega(g)$ se cumple:

   existen c, n_0 tal que $cg(n) \leq f_1(n)\ \forall n \geq n_0$ (1)

   Si $f_2 \in \Omega(h)$ se cumple:

   existen d, m_0 tal que $dh(n) \leq f_2(n)\ \forall n \geq m_0$ (2)

   de (1) y (2) obtenemos:

   $cdg(n)h(n) \leq f_1(n)f_2(n)\ \forall n \geq max\{n_0, m_0\}$

   entonces existen $e = cd, l_0 = max\{n_0, m_0\}$ y se cumple:

   $eg(n)h(n) \leq f_1(n)f_2(n)\ \forall n \geq l_0$

   $\Rightarrow f_1(n)f_2(n) \in \Omega(gh)$

8. Si existe $lim_{n\to\infty}\frac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:

   - *Si* $k \neq 0$ y $k < \infty \Rightarrow \Omega(f) = \Omega(g)$
   - *Si* $k = 0 \Rightarrow g \in \Omega(f)$, es decir. $\Omega(g) \subset \Omega(f)$, pero sin embargo se verifica que $f \notin \Omega(g)$

o notación

Esta es una acotación por arriba solamente de una función, que se define de la siguiente forma:

$o(g(n)) = \{f(n)/\ \forall\ c > 0, \exists\ n_0 > 0\ tal\ que\ 0 \leq f(n) < cg(n)\ \forall n \geq n_0\}$

Donde $f(x) = o(g(x))$ se puede interpretar que f crece mas despacio que g cuando x es muy grande.

ω notación

Esta es una acotación por abajo solamente de una función, que se define de la siguiente forma:

$\omega(g(n)) = \{f(n)/\ \forall\ c > 0,\ n_0 > 0\ tal\ que\ 0 \leq cg(n)\ < f(n)\ \forall n \geq n_0\}$

2.4.2. Propiedades

Algunas de las propiedades de las funciones de complejidad temporal son las siguientes:

- Transitiva:
  - $f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \rightarrow f(n) = \Theta(h(n))$
  - $f(n) = \mathcal{O}(g(n)) \wedge g(n) = \mathcal{O}(h(n)) \rightarrow f(n) = \mathcal{O}(h(n))$
  - $f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \rightarrow f(n) = \Omega(h(n))$
  - $f(n) = o(g(n)) \wedge g(n) = o(h(n)) \rightarrow f(n) = o(h(n))$
  - $f(n) = \omega(g(n)) \wedge g(n) = \omega(h(n)) \rightarrow f(n) = \omega(h(n))$
- Reflexiva:
  - $f(n) = \Theta(f(n))$
  - $f(n) = \mathcal{O}(f(n))$
  - $f(n) = \Omega(f(n))$
- Simétrica
  - $f(n) = \Theta(g(n))$ sí y solo sí $g(n) = \Theta(f(n))$
- Simétrica Transpuesta

- $f(n) = \mathcal{O}(g(n))$ sí y solo sí $g(n) = \Omega(f(n))$
- $f(n) = o(g(n))$ sí y solo sí $g(n) = \omega(f(n))$

2.4.3. Recurrencias

Existen algoritmos que son recursivos, de la forma general $T(n) = E(n)$, donde en la expresión $E(n)$ aparece la propia función $T(n)$, para poder analizar este tipo de algoritmos se debe de encontrar una ecuación no recursiva de $T(n)$. Algunas técnicas para hallar las ecuaciones no recursivas son:

- Método del teorema maestro
- Método de la ecuación característica

Método del teorema maestro

Se aplica en casos como:

$$T(n) = \begin{cases} 5 & \text{if } n = 0 \\ 9T(n) & \text{if } n \neq 0 \end{cases}$$

Es importante identificar:

- La cantidad de llamadas recursivas
- El cociente en el que se divide el tamaño de las instancias
- la sobrecarga extra a las llamadas recursivas

Teorema

Sean $a \geq 1, b > 1$ constantes, $f(n)$ una función y $T(n)$ una recurrencia definida sobre los enteros no negativos de la forma $T(n) = aT(n/b) + f(n)$, donde n/b puede interpretarse como [n/b]. Entonces:

- Si $f(n) \in \mathcal{O}(n^{log_b a-\epsilon})$ para algún $\epsilon > 0$ entonces $T(n) \in \Theta(n^{log_b a})$.
- Si $f(n) \in \Theta(n^{log_b a})$ entonces $T(n) \in \Theta(n^{log_b a} lgn)$.
- Si $f(n) \in \Omega(n^{log_b a+\epsilon})$ para algún $\epsilon > 0$ y satisface $af(n/b) \leq cf(n)$ para alguna constante $c < 1$ y n suficientemente grande entonces $T(n) \in \Theta(f(n))$.

Ejemplos:

- Si $T(n) = 9T(n/3) + n$ entonces $a = 9, b = 3$, se aplica el caso 1 con $\epsilon = 1$ y $T(n) \in \Theta(n^2)$.
- Si $T(n) = T(2n/3) + 1$ entonces $a = 1, b = 3/2$, se aplica el caso 2 y $T(n) = \Theta(logn)$.
- Si $T(n) = 3T(n/4) + nlogn$ entonces $a = 3$, $b = 4$, $f(n) = nlogn$, como $f(n) \in \Omega(n^{log_4 3+0,2})$ y Para un n suficientemente grande, $af(n/b) = 3(n/4)log(n/4) \leq (3/4)nlogn = cf(n)$, *con* $c = 3/4$, por lo que se aplica el caso 3 y $T(n) \in \Theta(nlogn)$.

Método de la ecuación característica

Se aplica a ciertas recurrencias lineales con coeficientes constantes como:

$$T(n) = \begin{cases} 5 & \text{if } n = 0 \\ 10 & \text{if } n = 1 \\ 5T(n-1) + 8T(n-2) + 2n & \text{if } n > 1 \end{cases}$$

En general sirve para recurrencias de la forma:

$T(n) = a_1 T(n-1) + a_2 T(n-2) + ... + a_k T(n-k) + b^n p(n)$

donde $a_i, 1 \leq i \leq k$, b son constantes y $p(n)$ es un polinomio en n de grado s.

Se resuelve siguiendo el siguiente procedimiento:

- Hallar las raíces no nulas de la ecuación característica:

  $(x^k - a_1x^{k-1} - a_2x^{k-2} - ... - a_k)(x-b)^{s+1} = 0$

  Raices: $r_i, 1 \leq i \leq k$, cada una con multiplicidad m_i.

- Las soluciones son de la forma de combinaciones lineales de estas raices de acuerdo a su multiplicidad.

  $T(n) = \sum_{i=1}^{l} \sum_{j=1}^{m_j} c_{ij} n^{j-1} r_i^n$

- Si se necesita, se encuentran valores para las constantes $c_{ij}, 1 \leq i \leq l, 0 \leq j \leq m_i - 1$ y $d_i, 0 \leq i \leq s-1$ según la recurrencia original y las condiciones iniciales (valores de la recurrencia para n=0,1,...)

Ejemplo:

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2T(n-1) + 1 & \text{if } n > 0 \end{cases}$$

- Si $b = 1$ y $p(n) = 1$ de grado 0, la ecuación característica es:

  $(x-2)(x-1)^{0+1} = 0$, con $r_1 = 2$,$m_1 = 1$ y $r_2 = 1$,$m_2 = 1$.

- La Solución general es de la forma $T(n) = c_{11}2^n + c_{21}1^n$.

- A partir de las condiciones iniciales se encuentra:

  $c_{11} + c_{21} = 0$ de $n = 0$ $2c_{11} + c_{21} = 1$ de $n = 1$

  de donde $c_{11} = 1$ y $c_{21} = -1$.

- La solución es $T(n) = 2^n - 1$

Capítulo 3

Aplicación: Algoritmo de PRIM y KRUSKAL

Para el desarrollo de los Algoritmos PRIM y KRUSKAL, se utilizó la Teoría de grafos y se aplica en el modelamiento de problemas de sistemas reales, en el cual los pesos y los costos son asociados a los arcos del grafo, así tenemos su utilización para resolver:

- En diseño de redes de transporte, donde los pesos pueden representar distancias asociadas a la interconexión entre un lugar y otro.
- En diseño de redes de telecomunicaciones, donde los pesos podrían representar distancias asociadas a la interconexión entre un par de servidores.
- En sistemas distribuidos, donde los pesos representarían el tráfico de los sistemas.
- En un mapa de una línea aérea, donde los arcos del grafo representan las rutas de vuelo y los pesos representan distancias o tarifas.

- En un circuito eléctrico, donde los arcos representan los cables y los pesos están asociados a la longitud de los cables o los costos de los cables.

La solución a cada uno de los problemas mencionados, consiste en hallar la forma de interrelacionar todos los vértices del modelo al menor costo, y en el grupo de algoritmos que solucionan este tipo de problemas tenemos a: PRIM y KRUSKAL.

Estos algoritmos generan una solución óptima, que es un árbol minimal o maximal, que es un árbol que contiene todos los vértices del grafo, con los arcos cuya suma total de pesos sea mínima o máxima según corresponda.

3.1. Introducción a la Teoría de Grafos [2]

La Teoría de Grafos se inicia con un trabajo del Leonhard Euler(1707-1783), en el año de 1736, en el cual utiliza la Teoría de Grafos para modelar el problema conocido como Los siete puentes de Konigsberg y luego la posterior solución mediante el planteamiento de un teorema para solucionarlo.

3.1.1. Grafo

Sea V un conjunto no vacío finito, y sea $E \subset V$xV, El par (V, E) se llama un grafo dirigido en V, ó un dígrafo en V, donde V es un conjunto de vértices, ó nodos, y E es un conjunto de arcos ó aristas dirigidos. A este grafo se denota como $G = (V, E)$.

Ejemplo: Sea $V = \{v_1, v_2, v_3, v_4, v_5\}$ el conjunto de vértices

Sea $E = \{v_1v_2, v_1v_3, v_1v_4, v_2v_4, v_2v_5\}$ el conjunto de arcos.

Se puede definir el grafo $G = (V, E)$ con dichos conjuntos.

Representación gráfica

Un grafo se representa mediante un diagrama en el cual a cada vértice le corresponde un punto y si dos vértices son adyacentes se unen sus puntos correspondientes mediante una línea o curva.

Ejemplo del grafo definido anteriormente:

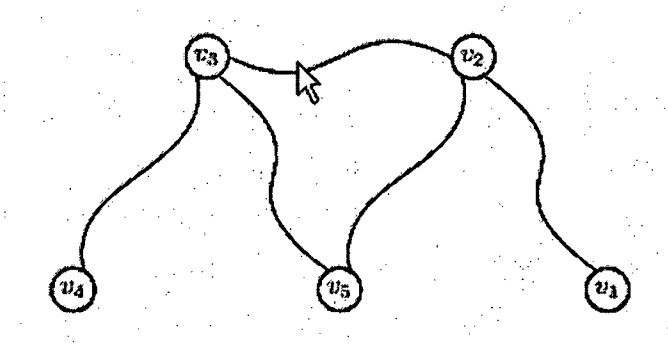


Figura 3.1: Representación gráfica de un grafo

Cuando no existe interés acerca de la dirección de los arcos, El conjunto E es un conjunto de pares sin orden de elementos pertenecientes a V, y a G se le llama un grafo no dirigido.

Ejemplo de grafo no dirigido ver figura 3.1.

Sea $G = (V, E)$ dirigido ó sin dirección, se llamará V el conjunto de vértices de G y a E el conjunto de arcos ó aristas de G.

En general si un grafo G no está definido como dirigido ó sin dirección, se asume que el grafo es sin dirección, si el grafo es dirigido se llamará grafo dirigido o dígrafo. Ejemplo de dígrafo o grafo dirigido:

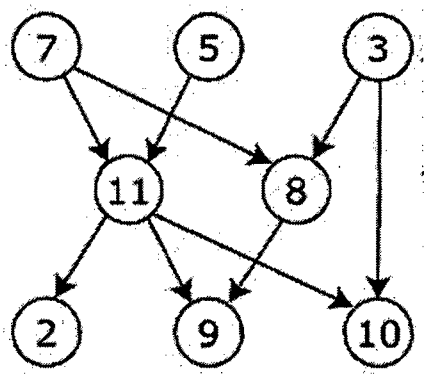


Figura 3.2: Representación gráfica de un grafo dirigido o dígrafo

También si el grafo no contiene lazos se le llamará libre de lazos o simple. Ejemplo de grafo que no contiene lazos ver figura 3.2.

Camino

Sea $G = (V, E)$ un grafo no dirigido ó sin dirección, entonces $x - y$ se llamará un camino en G si existe una secuencia alternada finita

$x = x_0, e_1, x_1, e_2, x_2, e_3, ..., e_{n-1}, x_{n-1}, e_n, x_n = y$

de vértices y arcos de G, empezando en el vértice x y finalizando en el vértice y, incluyendo los n arcos $e_i = x_{i-1}, x_i$, donde $1 \leq i \leq n$, además que ningún vértice ocurra más de una vez.

Ejemplo.

Un camino $v_4 - v_2$ en el grafo de la figura 3.1, sería el siguiente: $v_4, v_4v_3, v_3, v_3v_5, v_5, v_5v_2, v_2$

SubGrafo

Sea $G = (V, E)$ un grafo dirigido ó sin dirección, entonces $G_1 = (V_1, E_1)$ se llamará un subgrafo de G si $\phi \neq V_1 \subseteq V$ y $E_1 \subseteq E$, donde cada arco en E_1 es incidente con vértices en V_1.

Si $V_1 = V$, entonces G_1 se llamará un subgrafo de expansión de G.

Ejemplo de subgrafos:

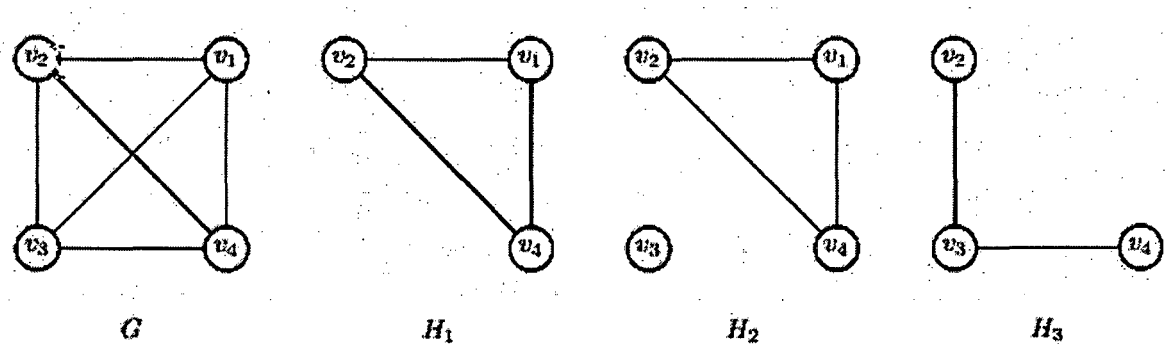


Figura 3.3: Representación de subgrafos

En la figura se tiene que H_1, H_2 yH_3 son subgrafos de G.

Grafo Conexo

Sea $G = (V, E)$ un grafo no dirigido ó sin dirección, se llamará a G conexo si existe un camino entre cualquier par de vértices distintos de G.

Ejemplo de grafo conexo es el grafo representado en la figura 3.1.

Un grafo que no es conexo se llamara disconexo.

Ejemplo de grafo disconexo:

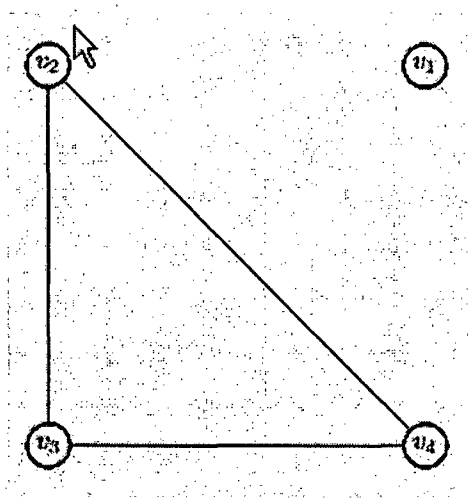


Figura 3.4: Representación de un grafo disconexo

3.1.2. Árboles

Sea $G = (V, E)$ un grafo no dirigido ó sin dirección sin lazos, G es un árbol si es conexo y no contiene ciclos.

Ejemplo de árbol:

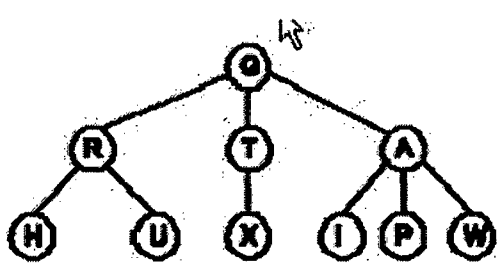


Figura 3.5: Representación de un arbol

Un árbol de expansión de un grafo conexo es un subgrafo de expansión que también es un árbol.

Ejemplo de árbol de expansión:

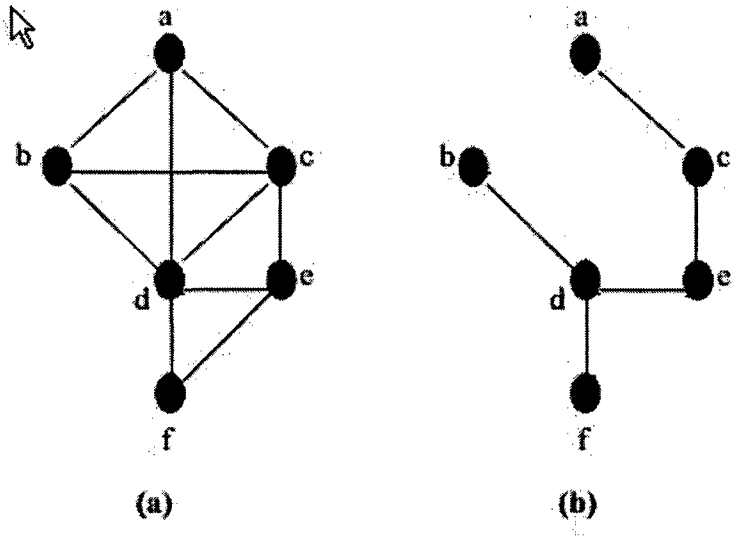


Figura 3.6: Representación de un arbol de expansión

En la figura 3.6 el árbol (b) es un árbol de expansión del grafo conexo (a).

Teoremas de Caracterización de árboles

Teorema.- G es un árbol si y solo si entre cada dos vértices existe sólo un camino(sin aristas respetidas).

Teorema.- G es un árbol si y solo si es conexo y el número de aristas es igual al de vértices menos 1.

Teorema.- Un grafo G es un árbol si y solo si es conexo y tiene la propiedad de que al eliminar una arista cualquiera del grafo, este deja de ser conexo.

Teorema.- Un grafo G es un árbol si y solo si no tiene ciclos y, si añadimos una arista cualquiera, se forma un ciclo.

Árbol Maximimal, Árbol Minimal

Sea $G = (V, E)$ un grafo conexo sin dirección y sin lazos , si a cada arco se le asigna un número real positivo que será su peso correspondiente, se denomina grafo con pesos.

Si T es un árbol de expansión de G, será un árbol Minimal o Maximal si la suma de los pesos de los arcos de T es el mínimo o máximo respectivamente de todos los árboles de expansión de G.

3.2. Algoritmo de KRUSKAL

Este algoritmo fue desarrollado por Joseph Bernard Kruskal en 1956, es del tipo ávido o voraz (greedy). El algoritmo de Kruskal tiene como entrada un grafo conexo con sus correspondientes pesos en sus arcos y como salida un árbol maximal o minimal, donde el peso total de las aristas en el árbol es maximizada o minimizada según corresponda. También es factible aplicarlo sobre grafos no conexos, para lo cual se aplica el algoritmo a cada componente conexo del grafo que modela el problema.

Desde un enfoque sistémico, usando el modelo básico podemos representar a este algoritmo de esta forma:

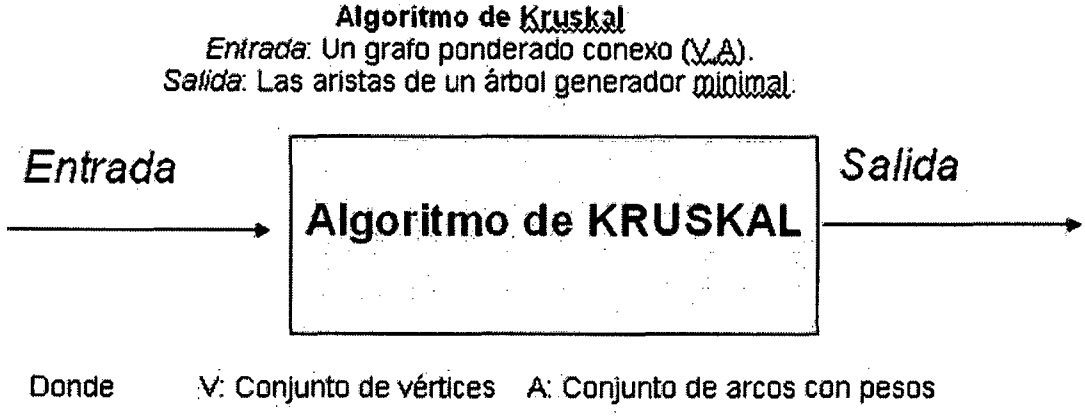


Figura 3.7: Kruskal

Algoritmo 1 Kruskal

Entrada: Grafo ponderado conexo $G(V, A)$

Salida: T=árbol de expansión mínimo del grafo $G(V, A)$

Ordenar los pesos de los vértices del Grafo G de menor a mayor

$n \leftarrow$ el número de vértices de G

$T \leftarrow \emptyset$

Inicializar n conjuntos, que cada uno contenga un elemento diferente de V

repetir

$e \leftarrow \{u, v\} \leftarrow$ el menor arco aun no considerado

$ucomp \leftarrow hallar(u)$

$vcomp \leftarrow hallar(v)$

si $ucomp \neq vcomp$ **entonces**

$unir(ucomp, vcomp)$

$T \leftarrow T \cup \{e\}$

fin si

hasta que T contenga $(n-1)$ arcos

La complejidad se puede analizar contemplando lo siguiente:

- Se tiene una complejidad de $O(aloga)$ para ordenar los arcos, el cual es equivalente a $O(aloga)$, como $n-1 \leq a \leq n(n-1)/2$
- Se tiene una complejidad de $O(n)$ al inicializar los n conjuntos disjuntos
- Se tiene una complejidad de $O((2a-1)log^*n$ para todas las búsquedas

Considerando que:

K llamadas a operaciones buscar el líder del conjunto que contiene a un vértice de conjuntos disjuntos de n elementos lleva un tiempo de $O(Klog^*n)$.

$log^*n \in O(logn)$, *pero* $logn \notin O(log^*n)$.

Por lo tanto podemos concluir que la complejidad del algoritmo de Kruskal es $O(alogn)$.

3.3. Algoritmo de PRIM

Este algoritmo fue diseñado en 1930 por el matemático Vojtech Jarnik y posteriormente de manera independiente por el científico computacional Robert C. Prim en 1957, y redescubierto por Dijkstra en 1959, razón por la cual muchos autores consideran a este algoritmo como: algoritmo DJP o algoritmo de Jarnik, es del tipo ávido o voraz (greedy).

El algoritmo de Prim tiene como entrada un grafo conexo con sus correspondientes pesos en sus arcos y como salida un árbol maximal o minimal, donde el peso total de las aristas en el árbol es maximizada o minimizada según corresponda. También es factible aplicarlo sobre grafos no conexos, para lo cual se aplica el algoritmo a cada componente conexo del grafo que modela el problema.

Desde un enfoque sistémico, usando el modelo básico podemos representar a este algoritmo de esta forma:

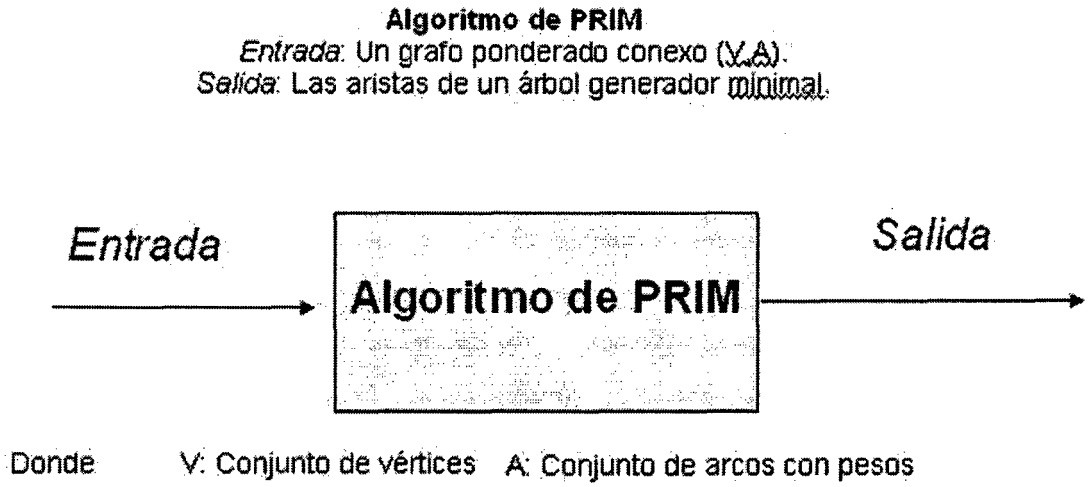


Figura 3.8: PRIM

Definamos:

1. B: Conjunto solución
2. cerca(i): Halla un nodo en el conjunto solución B que sea el mas cercano a i.
3. distanciamin(i): Halla la distancia del nodo i al nodo mas cercano.
4. L(i,j): matriz de pesos del grafo

Algoritmo de Prim:

Algoritmo 2 Prim

Entrada: Grafo ponderado conexo $G(V, A)$

Salida: T=árbol de expansión mínimo del grafo $G(V, A)$

$T \leftarrow \emptyset$

$B \leftarrow$ { Un vértice arbitrario de V }

mientras $B \neq V$ **hacer**

hallar $e = \{u, v\}$ de longitud mínima tal que $u \in B$ y $v \in V - B$

$T \leftarrow T \cup \{e\}$;

$B \leftarrow B \cup \{v\}$;

fin mientras

El bucle principal se ejecuta $n-1$ veces, en cada iteración cada bucle interior toma $O(n)$, por lo tanto el tiempo de ejecución del algoritmo de PRIM toma $O(n^2)$.

3.4. Comparación general

Tenemos las siguientes diferencias y similitudes:

- Ambos algoritmos son del tipo ávidos (greedy).
- El algoritmo de PRIM, va creando un solo árbol en todo el proceso de las iteraciones, en cambio el algoritmo de KRUSKAL, crea uno o varios árboles en cada iteración, finalizando con un solo árbol de expansión mínimo o máximo.
- El algoritmo de PRIM, se inicia desde un vértice cualquiera, en cambio el algoritmo de KRUSKAL, se inicia desde el arco o arista de menor o mayor peso según corresponda a una maximización o minimización.

- Las soluciones halladas por ambos algoritmos son óptimos e iguales siempre, es decir al final tienen el mismo peso total máximo o mínimo según corresponda a una maximización o minimización.

- Ambos algoritmos pueden generar mas de un árbol de expansión mínimo o máximo según corresponda a una maximización o minimización, en el caso los pesos fueran todos diferentes se genera como solución un único árbol, pero si hubiera al menos dos arcos con pesos iguales existe la posibilidad que se generen dos soluciones óptimas.

3.5. Comparación de la complejidad

La complejidad del algoritmo de PRIM es $O(n^2)$, en cambio la complejidad del algoritmo de KRUSKAL es $O(alogn)$, donde n es el número de vértices y a es el número de arcos.

Como $n-1 \leq a \leq \frac{n(n-1)}{2}$ se cumple, entonces:

- Si $a \approx n$ conviene utilizar Kruskal

- Si $a \approx n^2$ conviene utilizar Prim

Capítulo 4

Conclusiones

- La teoría de la complejidad permite analizar el tiempo computacional de un algoritmo, obteniéndose como resultado una función de complejidad temporal que dependerá de una variable relacionado al problema, en el caso del presente trabajo la función hallada depende del número de vértices del grafo que modela el problema a solucionar, como en el caso de los algoritmos de PRIM y KRUSKAL.

- Si tenemos definido un problema, que se puede modelar mediante un grafo conexo y la solución a dicho problema es mediante un árbol minimal o maximal, podemos usar los algoritmos de PRIM y KRUSKAL para hallar dichos árboles.

- Para escoger uno de los algoritmos de PRIM y KRUSKAL, se puede resumir en el siguiente análisis: Donde n es el número de vértices y a es el número de arcos.

  - Si $a \approx n$ conviene utilizar Kruskal
  - Si $a \approx n^2$ conviene utilizar Prim

Bibliografía

[1] LEISERSON C. CORMEN, T. and R RIVEST. *Introduction to Algorithms.* The MIT Press Mc Graw Hill. (p 1-16). London, England., 1990.

[2] Ralph P. Grimaldi. *DISCRETE AND COMBINATORIAL MATHEMATICS.* Addison-Wesley, 2004.

[3] Vallecillo Rosa, Guerequeta y Antonio. *Técnicas de Diseño de Algoritmos.* Servicio de Publicaciones de la Universidad de Málaga, 2000.

[4] Robert Sedgewick. *Algorithms.* Addison-Wesley, 1946.

[5] Herbert S. Wilf. *Algorithms and complexity.* University of Pennsylvania, 1994.