
ESTRUCTURAS COMPUTACIONALES AVANZADAS

Dr. en C. Luis Fernando Gutiérrez Marfileño

Universidad Autónoma de Aguascalientes

Centro de Ciencias Básicas

Departamento de Ciencias de la Computación

Academia de Inteligencia Artificial y Fundamentos Computacionales

8 de Agosto de 2022

Índice general

1. ALGORITMOS PARA OBJETOS COMBINATORIOS	2
1.1. Introducción	2
1.2. Algoritmos para generar permutaciones	9
1.2.1. Métodos basados en intercambios	10
1.2.2. Otros tipos de Algoritmos	15
1.3. Conjunto Potencia	17
1.4. Correspondencia entre el conjunto Potencia y cadenas binarias . .	19
1.5. Algoritmos para generar cadenas binarias	22
1.6. El conjunto de todos los subconjuntos	23
1.7. Correspondencia entre subconjuntos y cadenas binarias	24
1.8. Comparación de Algoritmos. Notación O grande	25
1.9. Problemas reales donde aplicar los algoritmos. Ejercicios	28
 2. GRAFOS, SUBGRAFOS Y ÁRBOLES	 29
2.1. Introducción	29
2.1.1. Concepto de grafo	29
2.1.2. Grafo conexo y no conexo	34
2.1.3. Grafos completos	36
2.1.4. Subgrafo	37
2.1.5. Componentes conexos de un grafo	39
2.1.6. Árbol y bosques	39
2.1.7. Diferentes formas de caracterizar el concepto de árbol . .	41
2.2. Guardando un grafo en una matriz de adyacencias	44
2.3. Cómo hacer un algoritmo para visitar cada nodo	47
2.3.1. Árboles Generales	50
2.3.2. Características y propiedades de los árboles	51
2.3.3. Longitud de camino interno y externo	52
2.4. Algoritmo para calcular todas las componentes conexas de un grafo	54
2.5. Algoritmo que encuentra el camino más corto	56
2.6. Saber si un grafo es un árbol o no	58
2.7. Árbol de expansión	59
2.7.1. Algoritmos para encontrar el árbol de expansión	59

2.7.2.	Algoritmo de Kruskal	59
2.7.3.	Algoritmo de Prim	60
2.7.4.	Aplicaciones reales donde encontrar el árbol	62
3.	GRAFOS BIPARTITOS	63
3.1.	Introducción	63
3.2.	Concepto de grafo bipartito	65
3.3.	Diferentes formas de caracterizar un grafo bipartito	68
3.4.	Concepto de pareo en un grafo	72
3.5.	Modelando problemas reales con grafos bipartitos	75
3.6.	Problemas de Pareo en grafos. Pareos perfectos y maximales . . .	82
3.7.	Aplicaciones	86

Índice de figuras

1.1. Objetos combinatorios	7
1.2. Algoritmos Combinatorios	8
1.3. Diagrama 1	10
1.4. Diagrama 2	11
1.5. Redes de permutaciones legales para 3 elementos	12
1.6. Algoritmo de Johnson Trotter	15
1.7. Permutaciones lexicográficas	16
1.8. Algoritmo generador de conteo binario (recursivo)	22
1.9. Procedimiento para convertir de decimal a binario	22
1.10. Solución de problemas	25
1.11. Implementación de algoritmos	25
2.1. Diagramas de grafos	30
2.2. Grafo 5,5	31
2.3. Multigrafo	33
2.4. Seudografo	33
2.5. Digrafo	34
2.6. Grafo conexo	34
2.7. Grafo no conexo	35
2.8. Grafos completos de 1 a 12 nodos	36
2.9. Subgrafos de un grafo	37
2.10. Grafo y subgrafo expandido	37
2.11. Grafo y subgrafo inducido	38
2.12. Grafo y subgrafo generador	38
2.13. Grafo y subgrafo recubridor	39
2.14. Diagrama de Venn	41
2.15. Notación identada	42
2.16. Grafo	42
2.17. Desplegado de la matriz resultante	45
2.18. Grafo resultante	45
2.19. Desplegado de la matriz resultante	45
2.20. Grafo resultante	46

2.21. Desplegado de la matriz resultante	46
2.22. Grafo resultante	46
2.23. $Recorrido_{amplitud} = \{12, 8, 17, 5, 9, 15\}$	47
2.24. $Recorrido_{preorden} = \{R, A, C, D, B, E, F\}$	48
2.25. $Recorrido_{inorden} = \{C, A, D, R, E, B, F\}$	48
2.26. $Recorrido_{postorden} = \{C, D, A, E, F, B, R\}$	49
2.27. Árbol general	51
2.28. Seudo código Warsall	54
2.29. Algoritmo de Dijkstra	57
3.1. Grafos de la misma <i>clase</i>	64
3.2. Grafo bipartito coloreado	64
3.3. Grafo bipartito completo $K_{2,4}$	64
3.4. Grafo bipartito en dos columnas	65
3.5. Grafo balanceado	65
3.6. Grafo cíclico	66
3.7. Hipergrafo	68
3.8. Modelo del espacio vectorial	69
3.9. Red de palabras (<i>word network</i>)	70
3.10. Grafo bipartito bicolor	71
3.11. Grafo unipartito resultante	71
3.12. NO es emparejamiento	73
3.13. SÍ es un emparejamiento	73
3.14. Grafo bipartito G_1	73
3.15. Grafo bipartito G_2	74
3.16. Arribos de aviones	75
3.17. Asignación actual de slots	76
3.18. Opciones de asignación	76
3.19. Grafo de reasignación	78
3.20. Problema de ruteo de vehículos	78
3.21. Distribución y abastecimiento de máquinas vending	79
3.22. Replanificación de las rutas iniciales	80
3.23. Grafo bipartito	80
3.24. Grafo de 3 emparejamientos perfectos	82
3.25. Grafo con 8 emparejamientos perfectos	83
3.26. Grafo con 36 emparejamientos perfectos	83
3.27. Emparejamiento máximo bipartito	85
3.28. Pseudocódigo Algoritmo de Gale-Shapley	88

Índice de tablas

1.1. Tiempos de permutaciones	9
1.2. Conjunto potencia	21
1.3. Conjunto potencia	24
3.1. Preferencias hombre	86
3.2. Preferencias mujer	87

Lista de Algoritmos

1.	Algoritmo de Kruskal	60
2.	Algoritmo de Prim	61

Unidad 1

ALGORITMOS PARA GENERAR OBJETOS COMBINATORIOS EN FORMA EXHAUSTIVA

“Comprender e interpretar objetos combinatorios y su utilidad en la solución de problemas; implementar y aplicar algoritmos que calculan objetos combinatorios; identificar objetos combinatorios.”

Objetivos específicos 1, 2 y 3

1.1. Introducción



La habilidad de *contar* es necesaria para sobrevivir, definir las porciones de alimentos para un grupo de personas, los días que transcurren para muchos eventos cíclicos naturales, la cantidad de cazadores necesarios para abatir una presa determinada, son ejemplos de que, desde su aparición, los seres humanos tuvieron que desarrollar esta habilidad (aunque, hoy se sabe que no es exclusiva de los humanos).

Es probable que éstas hayan sido las primeras operaciones matemáticas realizadas por nuestros antecesores.

Generalmente, los objetos contados forman conjuntos finitos, y hay muchas formas de crear tales *agrupaciones*.



Técnicas de conteo

Las técnicas de conteo son un conjunto de métodos de probabilidad para contar el número posible de arreglos dentro de un conjunto o varios conjuntos de objetos. Estas se usan cuando realizar las cuentas de forma manual se convierte en algo complicado debido a la gran cantidad de objetos y/o variables.

Estas técnicas son varias, pero las más importantes se fundamentan en dos principios básicos, que son el aditivo y el multiplicativo.

Principio Aditivo

Si un evento A puede ocurrir de m maneras diferentes, y otro evento B puede ocurrir de n maneras diferentes, además, si ocurre uno no puede ocurrir el otro, entonces, el evento A o el evento B , podrán ocurrir de $m + n$ formas.

Empleando un enfoque de conjuntos se dice que, un conjunto finito S se divide en partes S_1, \dots, S_k si las partes están desunidas y su unión es S .

Formalmente, sí:

$$S_i \cap S_j = \emptyset \quad \text{para } i \neq j \quad \text{y} \quad S_1 \cup S_2 \cup \dots \cup S_k = S$$

Entonces:

$$|S| = |S_1| + |S_2| + \dots + |S_k|$$

Por tanto la *cardinalidad*¹ del conjunto S es la suma de todas las maneras en las que éste puede ocurrir.

Principio Multiplicativo

Si un evento A se puede realizar de m formas diferentes y luego se puede realizar otro evento B de n formas diferentes, el número total de formas en que ambos pueden ocurrir es igual a $m \times n$.

Empleando un enfoque de conjuntos se dice que, si S es un conjunto finito que es el producto de S_1, \dots, S_k .

Formalmente, sí:

$$S = S_1 \times S_2 \times \dots \times S_k$$

Entonces:

$$|S| = |S_1| \times |S_2| \times \dots \times |S_k|$$

Por tanto la cardinalidad del conjunto S es el producto de todas las maneras en las que éste puede ocurrir.

El término **combinatoria**, tal como se usa actualmente, fue introducido en el trabajo de [Wilhelm Leibnitz, 1666] y se define como:

¹Cardinalidad: es la cantidad de elementos que posee un conjunto.

**Definición (► Combinatoria)**

Es la parte de las matemáticas que estudia el número de posibilidades de ordenación, selección e intercambio de los elementos de un conjunto, es decir, las combinaciones, variaciones y permutaciones.

[DRALE]

A las operaciones que se realizan para manipular estas agrupaciones se les denominan *conteo*² y *enumeración*³ de los elementos en un conjunto finito.

Existen distintas formas de realizar estas agrupaciones (también llamadas *objetos combinatorios*), ya sea que se pueda considerar la población completa del conjunto (m) o sólo una muestra (n), que se repitan sus elementos o no, o si influye o no el orden de colocación de dichos elementos.

Variaciones

Genéricamente, *variar* es: hacer que una cosa sea diferente en algo de lo que antes era [DRALE]. Matemáticamente, las variaciones son aquellas formas de agrupar los elementos de un conjunto teniendo en cuenta que:

- Se permite que se repitan los elementos y se puede hacer tantas veces como elementos tenga la agrupación.
- Influye el orden en que se colocan.

Existe dos tipos:

1. **Variaciones sin repetición.** Las variaciones sin repetición de m elementos tomados de n en n se definen como las distintas agrupaciones formadas con n elementos distintos, eligiéndolos de entre los m elementos disponibles, considerando una variación distinta a otra tanto si difieren en algún elemento como si están situados en distinto orden, el número de variaciones que se pueden construir se calcula mediante la fórmula:

$$V(m, n) = m * (m - 1) * (m - 2) \dots (\text{hasta tener } n \text{ factores}) \quad (1.1)$$

2. **Variaciones con repetición.** Las variaciones con repetición de m elementos tomados de n en n se definen como las distintas agrupaciones formadas con n elementos que pueden repetirse, eligiéndolos de entre los m elementos disponibles, considerando una variación distinta a otra tanto si difieren

²Conteo: es dar la cantidad o el calculo de cosas considerándolas como unidades homogéneas.

³Enumeración: es la enunciación ordenada y sucesiva de los componentes de un conjunto.



en algún elemento como si están situados en distinto orden, el número de variaciones que se pueden construir se calcula mediante la fórmula:

$$VR(m, n) = m_1 * m_2 * \dots * m_n = m^n \quad (1.2)$$

Permutaciones

Genéricamente, *permutar* es: variar la disposición u orden en que estaban dos o más cosas [DRALE]. Matemáticamente, las permutaciones son un caso particular de las variaciones donde $m = n$ así, las permutaciones son aquellas formas de agrupar los elementos de un conjunto teniendo en cuenta que:

- Se toman todos los elementos disponibles.
- Influye el orden en que se colocan.

Existe dos tipos:

1. **Permutaciones SIN repetición.** Cuando todos los elementos disponibles son distintos, el número de permutaciones de n elementos que se pueden construir se calcula mediante la fórmula:

$$P(n) = V(n, n) = n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1 \quad (1.3)$$

El producto de los n primeros números naturales reciben el nombre de factorial de n y se expresa así: $n!$, entonces la ecuación 1.3 se puede expresar como:

$$P(n) = n! \quad (1.4)$$

2. **Permutaciones CON repetición.** Cuando en los m elementos existen elementos repetidos (un elemento aparece a veces, otro b veces, otro c veces, etc.) verificándose que $a + b + c + \dots = m$, las permutaciones que se obtienen reciben el nombre de permutaciones con repetición, el número de permutaciones repetidas que se pueden construir se calcula mediante la fórmula:

$$P(m)^{a,b,c} = \frac{m!}{a!b!c!} \quad (1.5)$$

Combinaciones

Genéricamente *combinar* es: unir cosas diversas de manera que formen un compuesto [DRALE]. Matemáticamente, las combinaciones son aquellas formas de agrupar los elementos de un conjunto teniendo en cuenta que:

- Se permite que se repitan los elementos, se puede hacer tantas veces como elementos tenga la agrupación.



- NO influye el orden en que se colocan.

Existen dos tipos:

1. **Combinaciones SIN repetición.** Las combinaciones sin repetición de m elementos tomados de n en n se definen como las distintas agrupaciones formadas con n elementos distintos, eligiéndolos de entre los m elementos disponibles, considerando una variación distinta a otra sólo si difieren en algún elemento, (no influye el orden de colocación de sus elementos), el número de combinaciones que se pueden construir se calcula mediante la fórmula:

$$C(m, n) = \frac{m * (m - 1) * (m - 2) * \dots * (m - n + 1)}{n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1} \quad (1.6)$$

Considerando las ecuaciones 1.1 y 1.4, la ecuación 1.6 se puede expresar como:

$$C(m, n) = \frac{V(m, n)}{n!} \quad (1.7)$$

Otra forma de expresar el número de combinaciones de m elementos tomados de n en n es mediante la siguiente expresión:

$$\binom{m}{n}$$

Esta expresión se conoce como número combinatorio y se lee m sobre n .

2. **Combinaciones CON repetición.** Las combinaciones con repetición de m elementos tomados de n en n (de orden n) se definen como las distintas agrupaciones de n elementos eligiéndolos de entre los m elementos disponibles, considerando una variación distinta a otra sólo si difieren en algún elemento, (no influye el orden de colocación de sus elementos), el número de combinaciones repetidas que se pueden construir se calcula mediante la fórmula:

$$CR(m, n) = \binom{m + n - 1}{n} \quad (1.8)$$

La Fig.1.1 muestra las relaciones entre los diferentes objetos combinatorios.

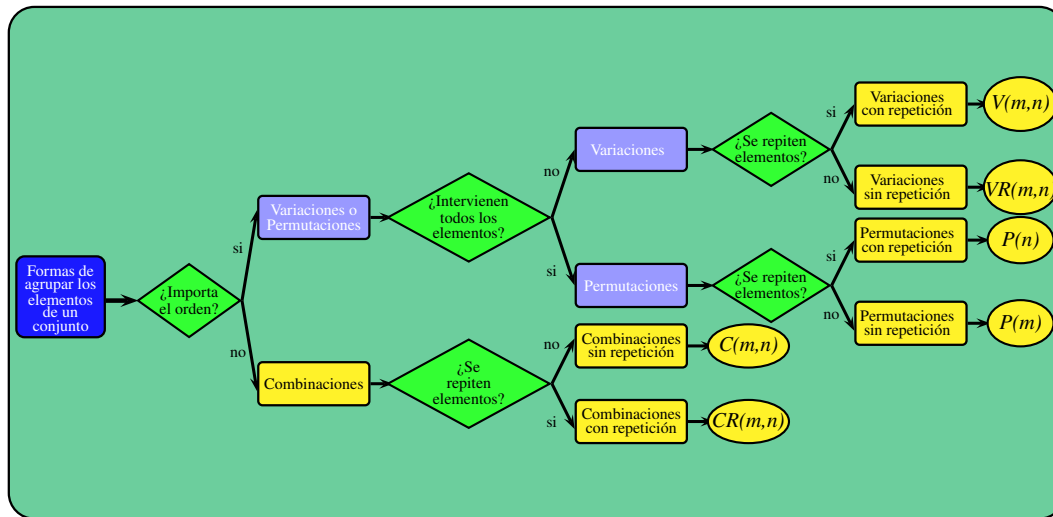


Figura 1.1: Objetos combinatorios

Cuando el número de elementos a agrupar es pequeño el cálculo de objetos combinatorios es relativamente fácil, pero si su número aumenta el problema empieza a complicarse.

Conforme aumenta el número de elementos a agrupar ocurre un fenómeno conocido como *explosión combinatoria*, en el cual la cantidad de agrupaciones a calcular crece exponencialmente.

Aunque los algoritmos para calcular estos objetos son aparentemente simples, a nivel humano cuando se requiere agrupar un número grande de elementos la cantidad de operaciones a realizar y el espacio que ocupan los resultados hace prácticamente imposible su conteo y enumeración.

Con la aparición de las computadoras digitales, a mediados del siglo XX, se aumentó enormemente la capacidad de realizar muchas operaciones y se ha ido ampliando su capacidad de memoria para almacenar datos, pero aún así, los desafíos combinatorios siguen siendo formidables. Por lo que, las formas para realizar cálculos de este tipo requieren de un cuidadoso análisis, debido a las limitaciones de los sistemas computacionales actuales.

Los algoritmos combinatorios se clasifican de acuerdo a su propósito en:

- **Generadores.** Construyen todas las estructuras de un tipo particular.
- **Enumeradores.** Calculan el número de diferentes estructuras de un tipo particular.
- **De búsqueda.** Encuentran al menos un ejemplo de una estructura de tipo particular o, comprueban que no existe.

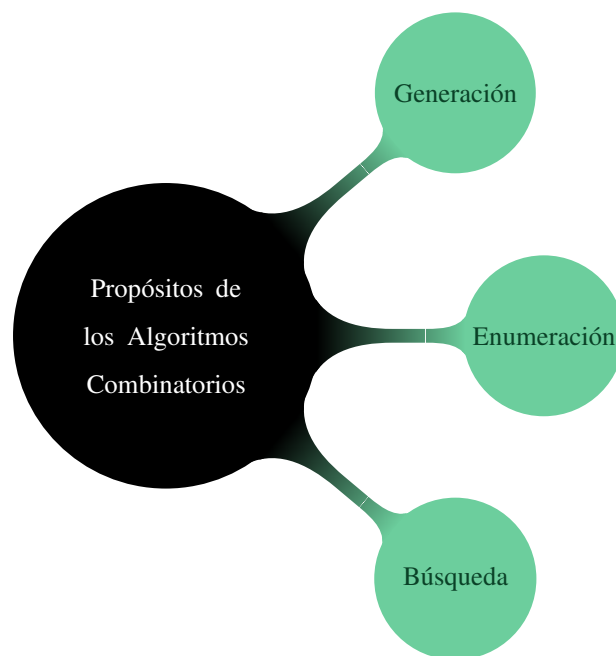


Figura 1.2: Algoritmos Combinatorios



1.2. Algoritmos para generar todas las permutaciones de n elementos



El estudio de los diversos métodos que se han propuesto para la generación de permutaciones mediante computadoras sigue siendo muy instructivo hoy porque ilustra muy bien la relación entre las operaciones de recuento, recursión e iteración [Sedgewick, 1977].

Los métodos de generación de permutaciones no solo ilustran problemas de programación en lenguajes de alto nivel (procedimentales); también ilustran problemas de implementación en lenguajes de bajo nivel (ensambladores).

Para muchas aplicaciones prácticas, la magnitud de $N!$ ha llevado al desarrollo de procedimientos de *búsqueda combinatoria* que son mucho más eficientes que la enumeración de permutación. Técnicas como la programación matemática y el retroceso (*backtraking*), se utilizan con regularidad para resolver problemas de optimización en aplicaciones industriales, y han llevado a la resolución de varios problemas difíciles en matemáticas combinatorias (por ejemplo, el problema de los cuatro colores).

N	$N!$	Tiempo
1	1	
2	2	
3	6	
4	24	
5	120	
6	720	
7	6,040	
8	40,320	
9	362,880	
10	3,628,800	3 seg.
11	39,916,800	40 seg.
12	479,001,600	8 min.
13	6,227,020,800	2 hrs.
14	87,178,291,200	1 día
15	1,307,674,368,000	2 sem.
16	20,922,789,888,000	8 meses
17	355,689,428,096,000	10 años

Tabla 1.1: Tiempo aproximado necesario para generar todas las permutaciones de N (considerando 1 microseg/permutación)



Se va a iniciar el estudio de los algoritmos generadores de permutaciones con algoritmos simples que generan permutaciones de una matriz mediante el intercambio sucesivo de elementos; todos estos algoritmos tienen una estructura de control común, y luego se estudiarán algunos algoritmos más antiguos, incluyendo algunos basados en operaciones elementales distintas de intercambios. Finalmente, se tratarán los problemas involucrados en la implementación, el análisis y la *optimización* de los mejores algoritmos.

1.2.1. Métodos basados en intercambios

Una forma natural de permutar una serie de elementos en una computadora es intercambiar dos de ellos. Los algoritmos de permutación más rápidos funcionan de esta manera: las permutaciones de $N!$ de los N elementos son producidas por una secuencia de intercambios $N! - 1$. Se usará la notación:

$$P[1] := P[2]$$

que significa *intercambiar* los contenidos de los elementos de la matriz $P[1]$ y $P[2]$. Esta instrucción proporciona ambas disposiciones de los elementos $P[1]$, $P[2]$ (es decir, la disposición antes del intercambio y la posterior).

Para $N = 3$, se pueden usar varias secuencias diferentes de cinco intercambios para generar las seis permutaciones, por ejemplo:

$$P[1] := P[2]$$

$$P[2] := P[3]$$

$$P[1] := P[2]$$

$$P[2] := P[3]$$

$$P[1] := P[2]$$

Si los contenidos iniciales de $P[1]$, $P[2]$ y $P[3]$ son $A B C$, estos cinco intercambios producirán las permutaciones BAC , BCA , CBA , CAB y ACB .

Será conveniente trabajar con una representación más compacta que describa estas secuencias de intercambio. Se va a considerar que los elementos pasan a través de *redes de permutación* que producen todas las permutaciones, éstas redes están compuestas de *módulos de intercambio* como el mostrado en la fig.1.3.

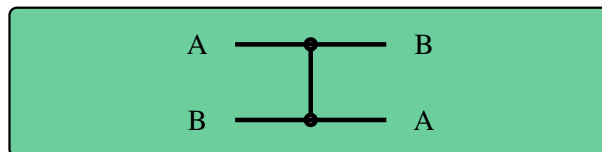


Figura 1.3: Diagrama 1

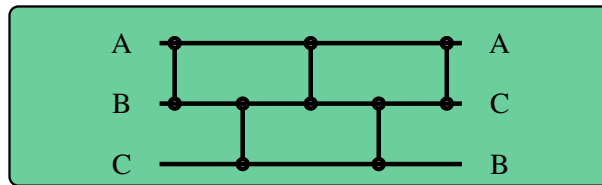


Figura 1.4: Diagrama 2

Que es en sí misma la red de permutación para $N = 2$, la red del Diagrama 2 implementa el intercambio de secuencia dada anteriormente para $N = 3$.

Los elementos pasan de derecha a izquierda, y una nueva permutación está disponible después de cada intercambio. Por supuesto, se debe estar seguro de que las permutaciones internas generadas son distintas.

Para $N = 3$ hay $3^5 = 243$ posibles redes con cinco módulos de intercambio, pero solo los doce que se muestran en la Fig.1.5 son *legales* (producen secuencias de permutaciones).

Se representan a menudo redes como en la Fig.1.5, concretamente dibujadas verticalmente, con elementos que pasan de arriba al fondo, y con las secuencias de permutaciones que se generan explícitamente escritas a la derecha.

Es fácil ver que para N más grande habrá un gran número de redes legales. Los métodos que ahora se analizarán muestran cómo construir sistemáticamente redes para una N arbitraria.

Por supuesto, se está más interesado en redes con una estructura suficientemente simple que su intercambio de las secuencias se pueden implementar convenientemente en una computadora.

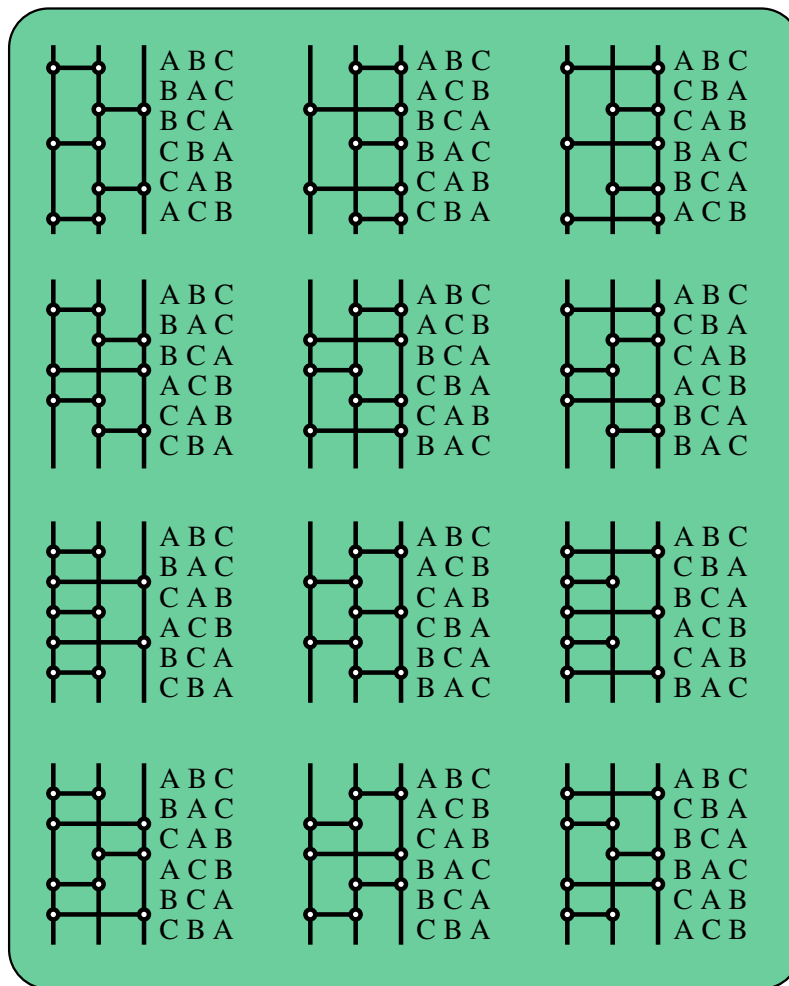


Figura 1.5: Redes de permutaciones legales para 3 elementos

Métodos recursivos

Se va a analizar una clase de métodos de generación de permutaciones que son muy simples cuando se expresan como programas recursivos.

Para generar todas las permutaciones de $P[1], \dots, P[N]$, se repiten N veces el paso: *primero generar todas las permutaciones de $P[1], \dots, P[N-1]$, luego intercambiar $P[N]$ con uno de los elementos $P[1], \dots, P[N-1]$.*

Como esto se repite, se pone un nuevo valor en $P[N]$ cada vez. Hay varios métodos que difieren en sus enfoques para llenar $P[N]$ con los N elementos originales.

El **Algoritmo de Heap** [Heap, 1963] es un generador de todas las permutaciones de N elementos, en este algoritmo se da un intercambio de un par de elementos mientras que los restantes $N-2$ elementos no se modifican.



Se tiene una permutación constituida por N elementos diferentes y se emplea un método sistemático para seleccionar en cada llamada un par de elementos para intercambiar con lo que se obtiene una permutación válida cada vez.

Inicialmente se declara una variable *contador* que inicia en 0 y después de realizar pasos en forma repetida termina en Num_ele .

Se utiliza el algoritmo para generar las $(N - 1)!$ permutaciones de los primeros $N - 1$ elementos, contiguos el último elemento a cada cual de estos.

Esto genera todas las permutaciones que terminan con el último elemento. Entonces si N es impar, se cambia el primer elemento y el último, mientras que si N es par se puede cambiar el i -ésimo elemento y el último (no hay ninguna diferencia entre N par e impar en la primera iteración).

Se incrementa en uno el *contador* y se repite en cada iteración, el algoritmo producirá todas las permutaciones que terminan con el elemento que acaba de ser movido a la última posición.

El siguiente procedimiento genera todas las permutaciones en forma recursiva de un arreglo de datos de longitud N .

```

1 procedure permutaciones(Numero_elem: int, Arreglo_perm: array)
2   if Numero_elem=1 then
3     salida (Arreglo_perm)
4   else
5     for contador:=0;contador<Num_elem-1;contador+=1 do
6       permutaciones(Num_elem-1,Arreglo_perm)
7       if Num_elem es par then
8         swap(Arreglo_perm[contador],Arreglo_perm[Num_elem-1])
9       else
10        swap(Arreglo_perm[0],Arreglo_perm[Num_elem-1])
11    permutaciones(Num_elem-1,Arreglo_perm)

```

Algoritmo recursivo de Heap

Intercambios adyacentes

Quizás el más prominente algoritmo de enumeración de permutaciones fue formulado por S.M. Johnson ([Johnson 1963]) y H.F. Trotter ([Trotter, 1959]), aparentemente en forma independiente.

Ellos descubrieron que es posible generar todas las $N!$ permutaciones de N elementos con $N! - 1$ intercambios de elementos adyacentes.

Cada permutación en la secuencia que genera difiere de la permutación previa al intercambiar dos elementos adyacentes de la secuencia.



De forma equivalente, este algoritmo encuentra un *ciclo hamiltoniano* en el permutahedro⁴.

Además de ser simple y computacionalmente eficiente, tiene la ventaja de que los cálculos posteriores en las permutaciones que genera pueden acelerarse porque estas permutaciones son muy similares a las demás.

Como describió Johnson, el algoritmo para generar la siguiente permutación a partir de una permutación dada Π , realiza los siguientes pasos:

- Para cada i de 1 a N , sea x_i la posición donde el valor i se coloca en la permutación Π . Si el orden de los números de 1 a $i - 1$ en la permutación Π define una permutación par, sea $y_i = x_i - 1$; de lo contrario, deje que $y_i = x_i + 1$.
- Encuentra el mayor número i para el que y_i define una posición válida en la permutación Π que contiene un número menor que i . Cambie los valores en las posiciones x_i y y_i .

Cuando no se puede encontrar el número i cumpliendo las condiciones del segundo paso del algoritmo, el algoritmo ha alcanzado la permutación final de la secuencia y finaliza. Este procedimiento puede implementarse en el tiempo $O(n)$ por permutación.

Debido a que este método genera permutaciones que alternan entre ser par e impar, puede modificarse fácilmente para generar solo las permutaciones pares o solo las permutaciones impares: para generar la siguiente permutación de la misma paridad a partir de una permutación dada, simplemente aplique el mismo procedimiento dos veces.

Conteo factorial

Emplea una implementación alternativa del método Johnson-Trotter, virtualmente todos los algoritmos de generación de permutaciones están basados en esquemas de *conteo factorial*.

Aunque aparecen en la literatura en una variedad de versiones, todos tienen la misma estructura de control que el programa de conteo elemental anterior.

Así como se han llamado a métodos como algoritmos recursivos porque generan todas las secuencias de $c[1], \dots, c[i - 1]$ incrementos intermedios de $c[i]$ para todo i se llamarán estos métodos como algoritmos iterativos porque iteran $c[i]$ a través de todos sus valores entre incrementos de $c[i + 1], \dots, c[N]$.

Algoritmos sin lazo interno

Una idea que atrajo mucha atención es que el algoritmo Johnson-Trotter podría mejorarse eliminando el lazo interno, lo cual tiene como propósitos principales:

⁴Permutahedro de orden n : en matemáticas, es un politopo $(n - 1)$ -dimensional incrustado en un espacio n -dimensional, cuyos vértices se forman permutando las coordenadas del vector.



```

i := 1;
loop while i < N: i := i + 1;
    c[i] := 1; d[i] := true; repeat;

c[i] := 0;
process;
loop:
i := N; x := 0;
if not d[i] then x := x + 1 endif;
d[i] := not d[i]; c[i] := 1; i := i - 1; repeat;
while i > 1;
    if d[i] then k := c[i] + x
    else k := i - c[i] + x endif;
P[k] := P[k + 1];
process;
c[i] := c[i + 1];
repeat:

```

Figura 1.6: Algoritmo de Johnson Trotter

- Encontrar el índice más alto cuyo contador no está agotado, para restablecer los contadores en los índices más grandes, y para calcular el desplazamiento x .
- Para reiniciar los otros contadores, se procede como se hizo al eliminar la recursión y reiniciarlos justo cuando estén incrementados, en lugar de esperar hasta que sean necesarios.
- Finalmente, en lugar de calcular una compensación *global* x , se puede mantener una matriz $x[i]$ que da la compensación actual en el nivel i : cuando $d[i]$ cambia de falso a verdadero, se incrementa $x[s[i]]$.

1.2.2. Otros tipos de Algoritmos

Ciclado anidado

Permutaciones diferentes de $P[1], \dots, P[N]$ se puede obtener girando la matriz, Aquí, se examinan los métodos de generación de permutación que se basan, únicamente en esta operación. Se supone que se tiene una operación primitiva $\text{rotate}(i)$. Que hace una rotación cíclica izquierda de los elementos $P[1], \dots, P[i]$, en otras palabras, $\text{rotar}(i)$ es equivalente a:

Algoritmos lexicográficos

Un ordenamiento particular de la permutación $N!$ de N elementos que es el ordenamiento *lexicográfico*, o alfabético.



El orden lexicográfico de las 6 permutaciones de A B C se muestra en la Fig.1.7.

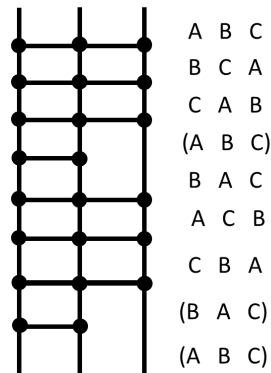


Figura 1.7: Permutaciones lexicográficas

La ordenación *inversa lexicográfica*, el resultado de leer la secuencia lexicográfica hacia atrás y las permutaciones de derecha a izquierda también son de interés.

La definición natural de estos ordenamientos ha significado que se han propuesto muchos algoritmos para la generación de permutaciones lexicográficas.

Tales algoritmos son intrínsecamente menos eficientes que los algoritmos de la sección anterior, porque a menudo deben usar más de un intercambio para pasar de una permutación a la siguiente en la secuencia.

Permutaciones aleatorias

Si N es tan grande que nunca se podría esperar generar todas las permutaciones de N elementos, es interesante estudiar métodos para generar x permutaciones *aleatorias* de N elementos.

Esto normalmente se hace estableciendo una correspondencia uno a uno entre permutación y un número aleatorio entre 0 y $N! - 1$.

Primero, notar que cada número entre 0 y $N! - 1$ se puede representar en un sistema mixto para corresponder a una matriz $c[N], c[N - 1], \dots, c[2]$ con $0 \leq c[i] \leq i - 1$ para $2 \leq i \leq N$. Por ejemplo, 1000 corresponde a 1 2 1 2 2 0 ya que $1000 = 6! + 2 * 5! + 4! + 2 * 3! + 2 * 2!$. Para $0 \leq n < N!$, tenemos $n = c[2] * 1! + c[3] * 2! + \dots + c[N] * (N - 1)!$.

Esta correspondencia se establece fácilmente a través de algoritmos de conversión radix estándar. Alternativamente, podríamos llenar la matriz poniendo un número *aleatorio* entre 0 hasta $i - 1$ en $c[i]$ para $2 \leq i \leq N$.



1.3. El conjunto Potencia es el conjunto de todos los posibles subconjuntos de un conjunto dado



ado un conjunto⁵ C , el conjunto potencia de C es otro conjunto formado exclusivamente por todos los subconjuntos de C .

Por ejemplo, dado el conjunto:

$$C = \{x, y, z\}$$

el conjunto potencia de C , representado como $\mathcal{P}(C)$, es:

$$\mathcal{P}(C) = [\emptyset, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, C]$$

Como se puede notar el conjunto resultante incluye tanto al conjunto vacío \emptyset como al conjunto original C .

Propiedades del Conjunto Potencia

- *Sobre el número de subconjuntos.* En primer lugar, el Álgebra de Conjuntos ha señalado que siendo el conjunto potencia, una colección abstracta conformada por los subconjuntos de un conjunto dado, se asume que siempre y en todo caso, **todo conjunto potencia contendrá al menos un elemento**, es decir, al menos un subconjunto de la colección en base a la cual se ha establecido.
- *Sobre el conjunto vacío.* Por otra parte, las distintas fuentes sobre Álgebra de Conjuntos coinciden en indicar que, debido a que el conjunto vacío cuenta con la propiedad de ser considerado, siempre y en todo momento, como un subconjunto de cualquier conjunto, entonces en el caso del conjunto potencia, esta colección siempre será anotada como uno de los subconjuntos de cualquier conjunto dado. En otras palabras, de acuerdo a lo que establece esta propiedad matemática, **el conjunto vacío (\emptyset) siempre estará presente en el conjunto potencia**, independientemente al conjunto dado. Para expresar matemáticamente esta propiedad, tenemos:

$$\emptyset \in \mathcal{P}(C) \text{ cualquiera que sea } C$$

⁵**Definición 1:** Un conjunto es una colección de objetos bien definidos y diferenciables entre sí que se llaman elementos.

Definición 2 : Se dice que un conjunto A está bien definido cuando, dado un elemento cualquiera x , es cierta una, y sólo una, de las siguientes proposiciones:

1. $x \in A$
2. $x \notin A$



- *Sobre el conjunto dado como subconjunto de sí mismo.* Así también, existe una ley matemática que indica que, siempre y bajo cualquier circunstancia, el conjunto es un subconjunto de sí mismo. En este sentido, se deriva otra propiedad matemática, en referencia al conjunto potencia, la cual indica que **en todo conjunto potencia aparecerá una vez el conjunto dado como subconjunto**, es decir, que todo conjunto, siendo el conjunto dado y sin importar cuál es, siempre aparecerá como elemento de su propio conjunto potencia. Esta propiedad puede ser expresada matemáticamente de la siguiente manera:

$$C \in \mathcal{P}(C) \text{ para cualquier } C$$

- *Sobre el cardinal del Conjunto potencia.* Finalmente, otra de las propiedades matemáticas que el Álgebra de conjuntos ha señalado como inherente al conjunto potencia está relacionada con el cardinal de esta colección, es decir con el número total de elementos que contiene. En este orden de ideas, esta disciplina matemática señala que, siempre y en todo momento, el cardinal del conjunto potencia será equivalente al resultado de una operación de potenciación, en donde la base equivalente a dos es elevada al cardinal del conjunto dado:

$$|\mathcal{P}(C)| = 2^{|C|}$$



1.4. Correspondencia entre el conjunto Potencia de un conjunto de n elementos y el conjunto de todas las cadenas binarias de tamaño n .

Como ya se señaló, el conjunto de todos los subconjuntos de un conjunto finito C es el conjunto potencia 2^C , y su cantidad de subconjuntos es igual a 2^n , donde n es el número de elementos que pertenecen al conjunto C , ahora:

¿Cómo formar el conjunto potencia de C cuando tiene éstos n elementos?

Para lograrlo suponer que el conjunto finito $C = \{a_1, a_2, \dots, a_n\}$ tiene n elementos, para formar el conjunto potencia 2^C , se asigna a cada subconjunto de C una cadena ordenada de n números binarios (0 y 1), indicando con el número 1 si el elemento forma parte de uno de los subconjuntos del Conjunto potencia, y 0 si el elemento no forma parte de dicho subconjunto.

Por ejemplo, la n -cadena ordenada de todos los ceros significa que el subconjunto es el conjunto vacío (es el conjunto que no tiene elementos), y la n -cadena ordenada de todos los unos es el mismo conjunto C (recordar que $\emptyset \subset C$ y $C \subset C$).

Para toda n -cadena ordenada de números binarios existe un único subconjunto del conjunto C , y viceversa.

Por lo tanto, el conjunto potencia del conjunto C , tiene tantos elementos como n -cadenas ordenadas se puedan formar.

Ahora, ¿Cuántas n -cadenas ordenadas de números binarios se pueden asignar a los subconjuntos que forman el conjunto potencia de C ?

Para construir las n -cadenas de números binarios a asignar a cada subconjunto que está en el conjunto potencia de C , se va a emplear el siguiente procedimiento: Para el primer elemento del conjunto C se tiene dos posibilidades (0 y 1) para el segundo elemento otras dos posibilidades, y así sucesivamente hasta llegar al n -ésimo elemento de C , el cual tiene 2^n posibilidades.

Es decir, para el primer elemento de C los números binarios 0 y 1 se alternan de mitad en mitad del total de subconjuntos que es 2^n .

Para el segundo elemento, los números binarios se alternan en la mitad de lo que se alternó en el primer elemento, y así sucesivamente hasta llegar al n -ésimo elemento en donde los números binarios se alternan de uno en uno.

Este procedimiento se implementa en el siguiente ejemplo.

**Ejemplo**

Determinar el Conjunto potencia del siguiente conjunto:

$$A = \{a, b, c, d, e\}$$

El conjunto potencia de A tiene $2^5 = 32$ elementos, para definirlo se creará un tabla que consta de $n+1$ columnas, en la primera fila se colocan los n elementos del conjunto A , y los subconjuntos se van a formar utilizando la n -cadena ordenada, de números binarios, la tabla consta de $2^n + 1$ filas (ver fig.1.2).



No.	a	b	c	d	e	Subconjuntos
1	0	0	0	0	0	$\{\emptyset\}$
2	0	0	0	0	1	$\{e\}$
3	0	0	0	1	0	$\{d\}$
4	0	0	0	1	1	$\{d,e\}$
5	0	0	1	0	0	$\{c\}$
6	0	0	1	0	1	$\{c,e\}$
7	0	0	1	1	0	$\{c,d\}$
8	0	0	1	1	1	$\{c,d,e\}$
9	0	1	0	0	0	$\{b\}$
10	0	1	0	0	1	$\{b,e\}$
11	0	1	0	1	0	$\{b,d\}$
12	0	1	0	1	1	$\{b,d,e\}$
13	0	1	1	0	0	$\{b,c\}$
14	0	1	1	0	1	$\{b,c,e\}$
15	0	1	1	1	0	$\{b,c,d\}$
16	0	1	1	1	1	$\{b,c,d,e\}$
17	1	0	0	0	0	$\{a\}$
18	1	0	0	0	1	$\{a,e\}$
19	1	0	0	1	0	$\{a,d\}$
20	1	0	0	1	1	$\{a,d,e\}$
21	1	0	1	0	0	$\{a,c\}$
22	1	0	1	0	1	$\{a,c,e\}$
23	1	0	1	1	0	$\{a,c,d\}$
24	1	0	1	1	1	$\{a,c,d,e\}$
25	1	1	0	0	0	$\{a,b\}$
26	1	1	0	0	1	$\{a,b,e\}$
27	1	1	0	1	0	$\{a,b,d\}$
28	1	1	0	1	1	$\{a,b,d,e\}$
29	1	1	1	0	0	$\{a,b,c\}$
30	1	1	1	0	1	$\{a,b,c,e\}$
31	1	1	1	1	0	$\{a,b,c,d\}$
32	1	1	1	1	1	$\{a,b,c,d,e\}$

Tabla 1.2: Conjunto potencia $\mathcal{P}(A)$



1.5. Algoritmos para generar todas las cadenas binarias de tamaño n .



Hay muchas formas de generar conteos binarios de tamaño n necesarios para determinar los subconjuntos del Conjunto potencia de un conjunto dado, a continuación dos de ellas.

```

Algoritmo gen_bin

Entradas: NUM_MAX: entero
Salidas: tabla bin: matriz
INICIO
    escribir: "Dame el numero hasta el que desees el conteo";
    leer: NUM_MAX
    for (CONT_DEC=0; CONT_DEC<NUM_MAX; CONT_DEC++)
    {
        escribir: "No_decimal", CONT_DEC+1;
        CONVERSOR(CONT_DEC);
    }
FIN
  
```

Figura 1.8: Algoritmo generador de conteo binario (recursivo)

El algoritmo mostrado en la fig. 1.8 señala la generación de un conteo en decimal mediante un proceso iterativo que inicia en 1 y termina en el numero máximo de conteo (NUM_MAX), después de generar cada numero se invoca un procedimiento para convertirlo en binario (fig. 1.9).

```

procedure CONVERSOR

void CONVERSOR(int CONT_BIN)
{
    int AUX;
    AUX ← CONT_BIN%2;
    tabla_bin ← AUX;
    escribir: " ", AUX;
    CONT_BIN ← CONT_BIN/2;
    CONVERSOR(CONT_BIN);
}
  
```

Figura 1.9: Procedimiento para convertir de decimal a binario

En dicho procedimiento se obtiene el residuo de la división y se va asignando a la tabla binaria bit por bit.



1.6. El conjunto de todos los subconjuntos de tamaño k de un conjunto dado



La cardinalidad de un conjunto -que suele denotarse por una doble barra sobre el nombre del conjunto- es la medida o cantidad de elementos que un conjunto tiene.

El conjunto del ejemplo anterior tiene seis elementos, entonces su cardinalidad es 6; eso se denota así:

$$\overline{\overline{A}} = 6$$

La cantidad o número de subconjuntos de un conjunto dado es siempre una potencia de 2.

Al conjunto de todos los posibles subconjuntos de un conjunto dado se le llama el Conjunto potencia de y se denota por

$$\mathcal{P}(A)$$

Como agrupar los subconjuntos en grupos k

Una característica común de todos los elementos de los subconjuntos (además de pertenecer al mismo conjunto) es el número de elementos que forman cada subconjunto.

Es decir, hay subconjuntos formados por 1 elemento, o por 2 elementos o,..., por n -elementos, la agrupación puede darse de la siguiente forma:



1.7. Correspondencia entre el conjunto de los subconjuntos k de un conjunto de n elementos, con las cadenas binarias de tamaño n con k unos.

Ejemplo

Para el conjunto:

$$A = \{a, b, c, d\}$$

Cuya cardinalidad es:

$$\overline{\overline{A}} = 4$$

Y su numero de subconjuntos es:

$$\mathcal{P} = 2^4 = 16$$

Y el numero de grupos de subconjuntos que pueden formarse es:

$$k = 4$$

Para generar el conjunto potencia tenemos:

No.	a	b	c	d	Subconjuntos	Rangos
1	0	0	0	0	$\{\emptyset\}$	k_0
2	0	0	0	1	$\{d\}$	k_1
3	0	0	1	0	$\{c\}$	k_1
4	0	0	1	1	$\{c, d\}$	k_2
5	0	1	0	0	$\{b\}$	k_1
6	0	1	0	1	$\{b, d\}$	k_2
7	0	1	1	0	$\{b, c\}$	k_2
8	0	1	1	1	$\{b, c, d\}$	k_3
9	1	0	0	0	$\{a\}$	k_1
10	1	0	0	1	$\{a, d\}$	k_2
11	1	0	1	0	$\{a, c\}$	k_2
12	1	0	1	1	$\{a, c, d\}$	k_3
13	1	1	0	0	$\{a, b\}$	k_2
14	1	1	0	1	$\{a, b, d\}$	k_3
15	1	1	1	0	$\{a, b, c\}$	k_3
16	1	1	1	1	$\{a, b, c, d\}$	k_4

Tabla 1.3: Conjunto potencia k_4



1.8. Comparación de Algoritmos.

Notación O grande

De acuerdo con lo que se ha visto, un problema puede ser resuelto mediante diferentes algoritmos.

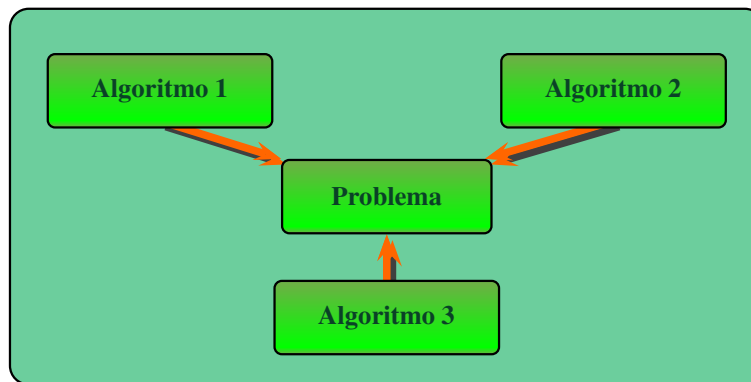


Figura 1.10: Solución de problemas

Por otra parte existen diferentes formas de crear algoritmos, a esta operación se le llama *implementación de algoritmos*.

Y cada una puede estar formada por un conjunto diferente de instrucciones.

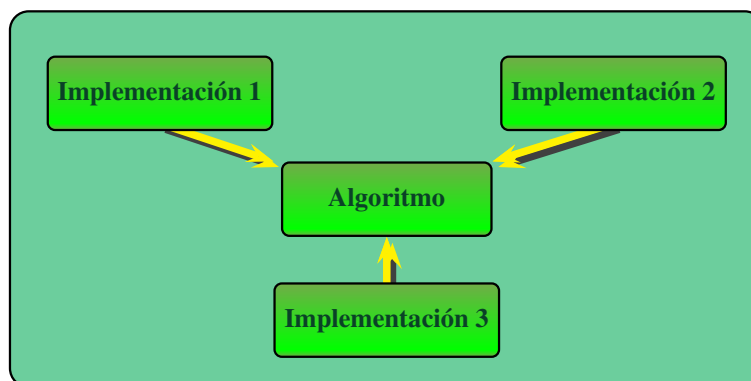


Figura 1.11: Implementación de algoritmos

El **análisis de algoritmos** pretende descubrir si estos son o no eficaces. Establece además una comparación entre los mismos con el fin de saber cuál es el más eficiente, aunque cada uno de los algoritmos de estudio sirva para resolver el mismo problema.



El *tiempo de ejecución* de un algoritmo depende de los datos de entrada, de la implementación del programa, del procesador y finalmente de la complejidad del algoritmo.

Sin embargo, se dice que *el tiempo que requiere un algoritmo para resolver un problema está en función del tamaño n del conjunto de datos para procesar*.

Hay dos tipos de estudios del tiempo que tarda un algoritmo en resolver un problema:

- En los estudios *a priori* se obtiene una medida teórica, es decir, una función que acota (por arriba o por abajo) el tiempo de ejecución del algoritmo para unos valores de entrada dados.
- En los estudios *a posteriori* se obtiene una medida real, es decir, el tiempo de ejecución del algoritmo para unos valores de entrada dados en determinada computadora.

Al tiempo que tarda un algoritmo para resolver un problema se le llama $T(n)$, donde n es el tamaño de los datos de entrada. Es decir, $T(n)$ está en función del tamaño de los datos de entrada.

El tamaño de la entrada es el número de componentes sobre los que se va a ejecutar el algoritmo.

El tiempo de ejecución $T(n)$ de un algoritmo para una entrada de tamaño n no debe medirse en segundos, milisegundos, etc., porque $T(n)$ debe ser independiente del tipo de computadora en el que se ejecuta.

En realidad lo que importa es la forma que tiene la función $T(n)$ para saber cómo cambia la medida del tiempo de ejecución en función del tamaño del problema, es decir, con $T(n)$ se puede saber si el tiempo aumenta poco o aumenta mucho cuando n crece.

La eficiencia de un algoritmo

Dado un algoritmo y dos implementaciones suyas I_1 e I_2 , que tardan $T_1(n)$ y $T_2(n)$ respectivamente, el **Principio de invarianza** afirma que existe una constante real $c > 0$ y un número natural n_0 tales que para todo $n = n_0$ se verifica que:

$$T_1(n) = cT_2(n)$$

Es decir, el tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa.

Esto significa que el tiempo para resolver un problema mediante un algoritmo depende de la naturaleza del algoritmo y no de la implementación del algoritmo.



Decimos entonces que el tiempo de ejecución de un algoritmo es asintóticamente del orden de $T(n)$ si existen dos constantes reales c y n_0 y una implementación I del algoritmo tales que el problema se resuelve en menos de $cT(n)$, para toda $n > n_0$.

Cuando se quiere comparar la eficiencia temporal de dos algoritmos, tiene mayor influencia el tipo de función que la constante c .

El tiempo de ejecución $T(n)$ de un algoritmo

Para medir $T(n)$ usamos el número de operaciones elementales. Una operación elemental puede ser:

- Una operación aritmética
- Una asignación a una variable
- Una llamada a una función
- El retorno de una función
- Comparaciones lógicas (con salto)
- El acceso a una estructura (arreglo, matriz, lista ligada, etc.)

Se le llama tiempo de ejecución, no al tiempo físico, sino al número de operaciones elementales que se llevan a cabo en el algoritmo.

Medidas asintóticas

Las cotas de complejidad, también llamadas medidas asintóticas sirven para clasificar funciones de tal forma que podamos compararlas.

Las medidas asintóticas permiten analizar qué tan rápido crece el tiempo de ejecución de un algoritmo cuando crece el tamaño de los datos de entrada, sin importar el lenguaje en el que esté implementado ni el tipo de máquina en la que se ejecute. Existen diversas notaciones asintóticas para medir la complejidad, una de las más comunes es la notación O grande.

$O(g(n))$ es el conjunto de todas las funciones f_i para las cuales existen constantes enteras positivas k y n_0 tales que para $n = n_0$ se cumple que:

$$f_i(n) = kg(n)$$

$kg(n)$ es una *cota superior* de toda f_i para $n = n_0$.

Cuando la función $T(n)$ está contenida en $O(g(n))$, entonces la función $cg(n)$ es una cota superior del tiempo de ejecución del algoritmo para alguna c y para toda $n = n_0$.

Lo que indica que dicho algoritmo nunca tardará más que: $kg(n)$. Recordar que el tiempo de ejecución es el número de operaciones elementales que se llevan a cabo y que n es el tamaño de los datos de entrada.



1.9. Problemas reales donde aplicar los algoritmos. Ejercicios

Ver sección de prácticas.

Unidad 2

GRAFOS, SUBGRAFOS Y ÁRBOLES

“Comprender e interpretar los conceptos de grafo, subgrafo y árbol, así como distinguir sus propiedades, implementar y aplicar algoritmos para recorrer grafos, conocer e identificar aplicaciones de la modelación de problemas con grafos y árboles.”


Objetivos específicos 1, 2 y 3

2.1. Introducción

Concepto de grafo, grafo conexo y no conexo, grafos completos, subgrafo, componentes conexas de un grafo, árbol y bosques.

Diferentes formas de caracterizar el concepto de árbol

2.1.1. Concepto de grafo

n **grafo** es un sistema matemático abstracto que permite representar relaciones binarias entre elementos de un conjunto. Los grafos permiten estudiar las interrelaciones entre unidades que interactúan unas con otras.

Para facilitar su estudio los grafos se van a representar mediante *diagramas*. A los cuales también se les llama grafos, aún cuando los términos y definiciones no estén limitados únicamente a los grafos que pueden representarse mediante diagramas.



Por lo que, prácticamente cualquier problema puede representarse mediante un grafo.

Su estudio proviene de una rama de la **Topología**¹ llamada **Teoría de Grafos** que es el estudio de estructuras matemáticas que se usan para modelar relaciones entre objetos de una colección.

Representación gráfica

Un grafo se representa mediante el diagrama (*drawing*) de un conjunto de objetos llamados *nodos* unidos por enlaces llamados *arcos*.

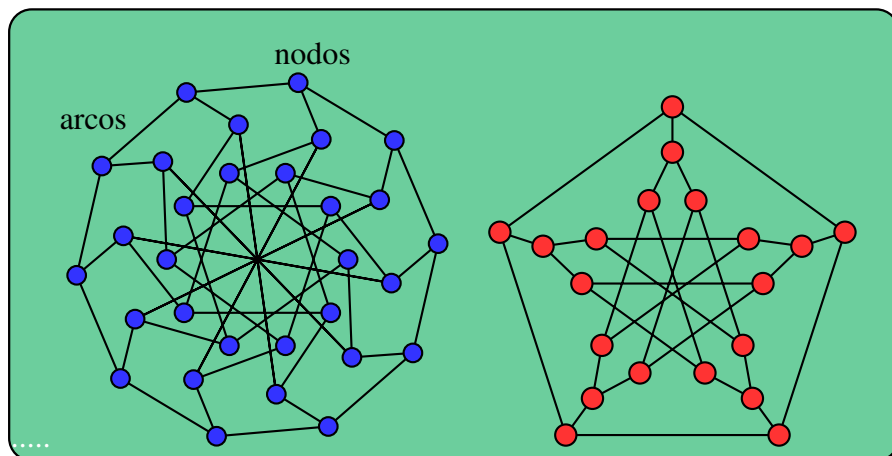


Figura 2.1: Diagramas de grafos

Matemáticamente

Un grafo G , es un par ordenado formado por un conjunto finito no vacío N , y un conjunto A , de pares no ordenados de elementos del mismo.

Donde:

- N es el conjunto de los nodos del grafo.
- A es el conjunto de los arcos del grafo.

Se usa la siguiente notación para designar al grafo cuyos conjuntos de nodos y arcos son, respectivamente N y A .

¹**Topología:** es la rama de las matemáticas dedicada al estudio las propiedades de los cuerpos geométricos que permanecen inalteradas por transformaciones continuas.



$$G = (N, A)$$

A cualquier arista de un grafo se le puede asociar un par de nodos del mismo. Si u y v son dos nodos de un grafo y la arista a está asociada con este par, escribiremos $a = uv$.

Ejemplo

Se tiene el siguiente grafo cuyos nodos son:

$$N = n_1, n_2, n_3, n_4, n_5$$

y sus arcos:

$$A = n_1n_2, n_1n_3, n_1n_4, n_2n_4, n_2n_5$$

entonces el grafo $G = (N, A)$ tiene a n_1, n_2, n_3, n_4 y n_5 como elementos unidos por $n_1n_2, n_1n_3, n_1n_4, n_2n_4$ y n_2n_5 , entonces su diagrama queda:

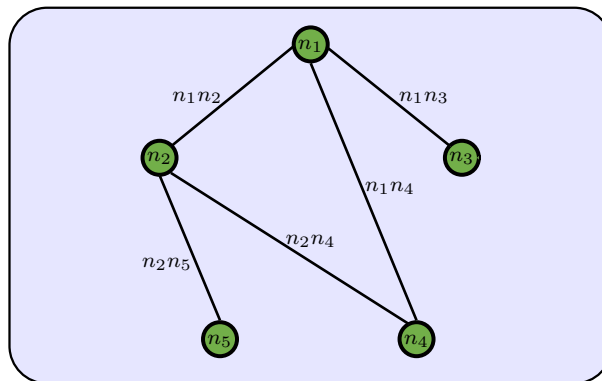


Figura 2.2: Grafo 5,5

Elementos de un grafo

A continuación se describen los elementos de un grafo.

Arcos

Son las líneas con las que se unen los nodos de un grafo y con la que se construyen también caminos. Si la arista carece de dirección se denota indistintamente $\{v, u\}$ o $\{u, v\}$, siendo v y u los nodos que une.

Si $\{u, v\}$ es un arco, a los nodos u y v se les denomina *extremos*.

- *Arcos Adyacentes*: Se dice que dos arcos son adyacentes si convergen en el mismo nodo.



- *Arcos Paralelas*: Se dice que dos arcos son paralelas si el nodo inicial y el final son el mismo.
- *Arcos Cíclicos*: Arco que parte de un nodo para entrar en el mismo.
- *Cruce*: Son dos arcos que cruzan en un punto.

Nodos

Son los elementos con los que esta conformado un grafo.

Se denomina *grado* de un nodo al número de arcos de las que es extremo. Se dice que un nodo es *par* o *impar* según lo sea su grado.

- *Nodos Adyacentes*: si tenemos un par de nodos de un grafo $\{v, u\} \in G$ y si tenemos un arista que los une, entonces v y u son nodos adyacentes y se dice que v es el nodo *inicial* y u el nodo *adyacente*.
- *Nodo Aislado*: Es un nodo de grado cero.
- *Nodo Terminal*: Es un nodo de grado 1.

Caminos

Sean los nodos $\{v, u\} \in G$, se dice que hay un camino en el grafo v a u si existe una sucesión finita no vacía de arcos $v, v_1, v_1, v_2, \dots, v_n, u$. En este caso

- Los nodos v y u se denominan *extremos del camino*.
- El número de arcos del camino se denomina *longitud* del camino.
- Si los nodos no se repiten el camino se dice propio o *simple*.
- Si hay un camino no simple entre 2 nodos, también habrá un camino simple entre ellos.
- Cuando los dos extremos de un camino son iguales, el camino se llama *circuito* o camino cerrado.
- Se denomina *ciclo* a un circuito simple
- Un nodo v se dice *accesible* desde el nodo u si existe un camino entre ellos. Todo nodo es accesible respecto a si mismo.



Clasificación de grafos

De acuerdo a su conformación tenemos los siguientes tipos de grafos.

Multigrafo

A los grafos en los que haya pares de nodos unidos por más de una arista se les denomina *multigrafos*.

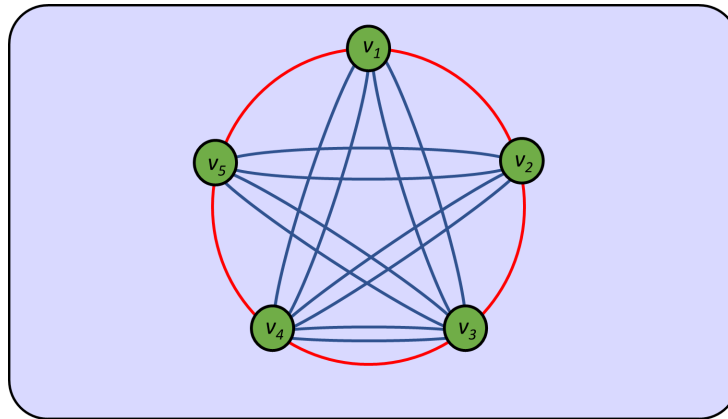


Figura 2.3: Multigrafo

Seudografo

A los grafos en los que existan arcos cuyos extremos coincidan (es decir, aquellos en los que existan arcos que unan nodos consigo mismos), se les denominan *seudografos* y a tales arcos se les denominan *bucles* o *lazos*.

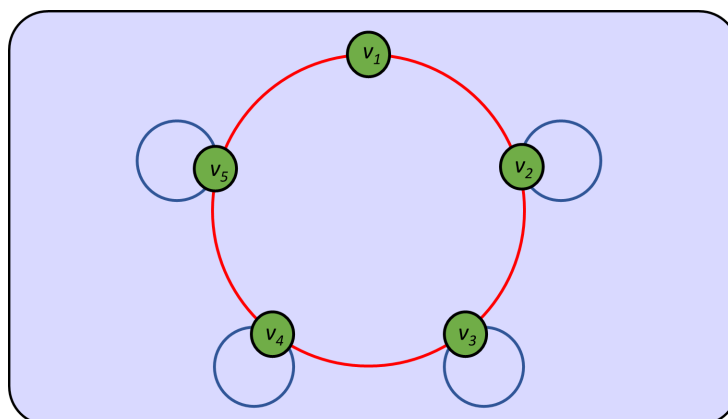


Figura 2.4: Seudografo

**Digrafo**

Es un grafo en el cual el conjunto de las arcos A está formado por pares ordenados del conjunto de nodos V . También se le denomina grafo dirigido.

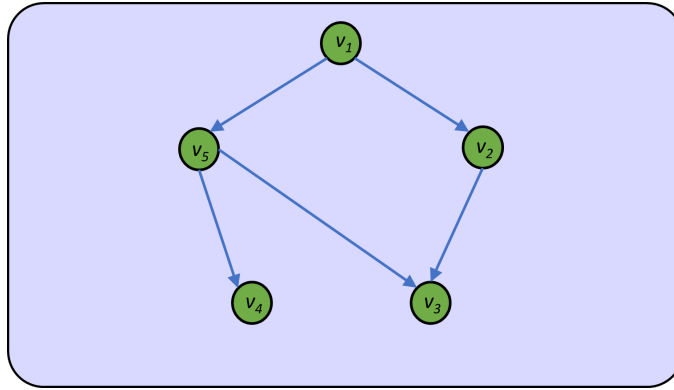


Figura 2.5: Digrafo

2.1.2. Grafo conexo y no conexo

Diremos que un grafo $G = (N, A)$ es conexo (o fuertemente conexo) si y sólo si, hay un camino entre cualquier par de nodos diferentes del grafo.

Además un grafo conexo es un grafo no-dirigido.

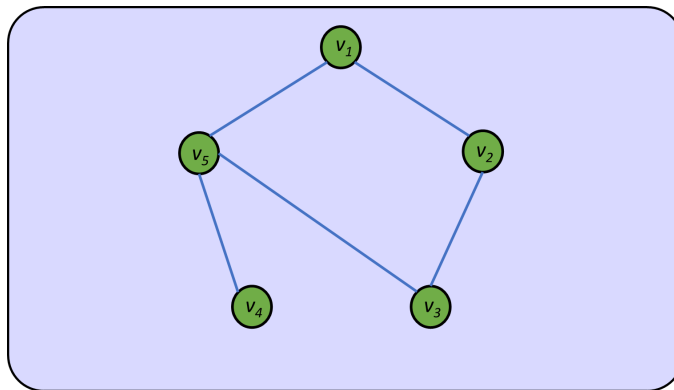


Figura 2.6: Grafo conexo

Proposición

Dado un grafo, la relación *estar conectado con* definida en el conjunto de sus nodos es una *relación de equivalencia*.



Para comprobar que exista una relación de equivalencia se verifica lo siguiente:

- **Reflexividad:** Sea u cualquier nodo de un grafo, entonces el camino m que conecta a u con u es:

$$\forall u \in V, uRu$$

La relación es reflexiva

- **Simetría:** Sean v y u dos elementos cualquiera del conjunto de nodos de un grafo entonces:

$$vRu \leftrightarrow \exists \mu = \langle v, u \rangle \leftrightarrow \exists m = \langle u, v \rangle \leftrightarrow uRv$$

Luego

$$\forall v, u \in V; vRu \rightarrow uRv$$

Es decir, la relación es simétrica

- **Transitividad:** Si v , u y w son tres nodos cualquiera de un grafo entonces:

$$\left. \begin{array}{l} uRv \leftrightarrow \exists \mu_1 = \langle u, v \rangle \\ vRw \leftrightarrow \exists \mu_2 = \langle v, w \rangle \end{array} \right\} \rightarrow \exists \mu = \langle u, w \rangle \leftrightarrow uRw$$

Bastaría con unir los caminos μ_1 y μ_2 , por lo tanto:

$$\forall u, v, w; uRv \wedge vRw \rightarrow uRw$$

Es decir, R es transitiva.

Un grafo no-conexo es aquel en el que al menos uno de sus nodos no está conectado con el resto de los demás.

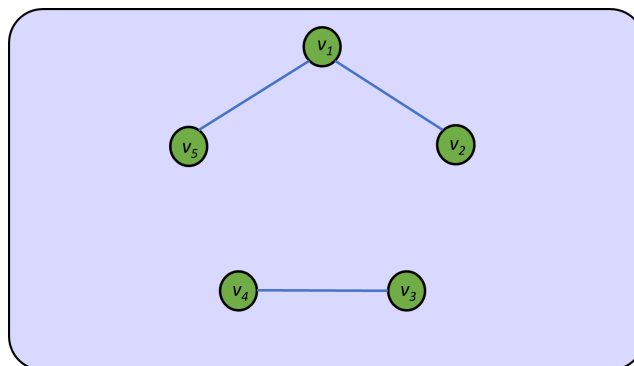


Figura 2.7: Grafo no conexo



2.1.3. Grafos completos

En teoría de grafos, un grafo completo es un grafo simple donde cada par de nodos está conectado por un arco.

Un grafo completo de n nodos tiene $n(n-1)/2$ arcos y se denotan por K_n .

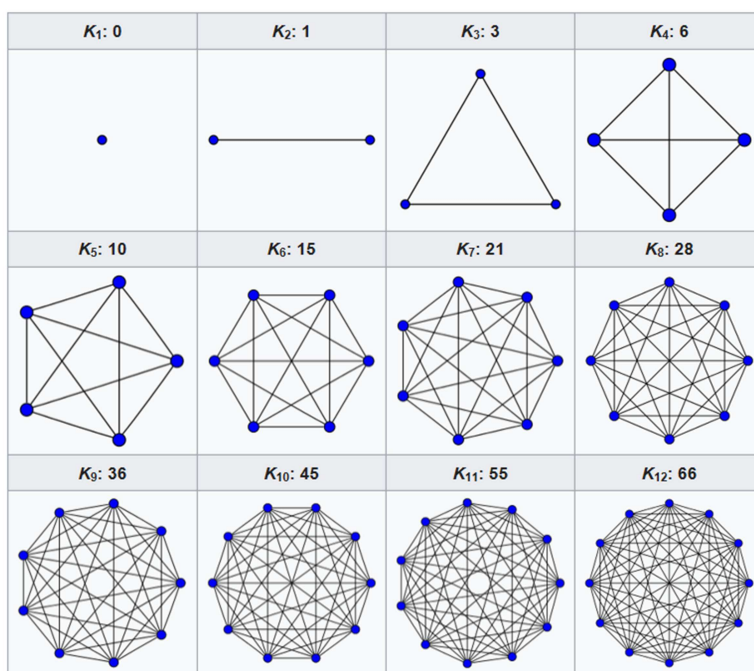


Figura 2.8: Grafos completos de 1 a 12 nodos

La familia de grafos K_n es la más simple de los grafos completos, aunque en el concepto general no excluye la presencia de multiarcos y lazos, haciendo que los grafos completos no sean necesariamente simples.

Esto produce caminos mínimos entre cualquier par de nodos aunque debe señalarse que para el caso de los arcos etiquetados la minimalidad de los caminos no correspondería necesariamente a arcos que unen el par de nodos sino posiblemente a caminos con más arcos en dependencia de la suma total de sus valores.

Matriz de adyacencia

Es una matriz cuadrada de orden $N \times N$ asociada a un grafo de orden N , donde sus filas y columnas se identifican con los nodos del grafo y en las celdas se indican la cantidad de arcos (o arcos salientes si es un dígrafo) a los nodos asignado a la fila y columnas en cuestión.

La matriz de adyacencia en la familia K_n es fácil de reconocer pues todos sus elementos tienen valor 1, excepto los de la diagonal principal que son 0.



2.1.4. Subgrafo

Definición:

Un subgrafo de un grafo $G = (N(G), A(G))$ es un grafo $H = (N(H), A(H))$ tal que $N(H) \subseteq N(G)$ y $A(H) \subseteq A(G)$.

Podría decirse que un subgrafo es un grafo dentro de otro grafo, este a su vez, es un supergrafo de su subgrafo.

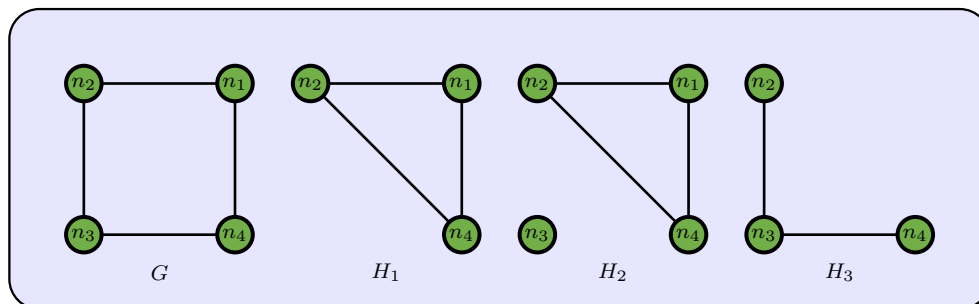


Figura 2.9: Subgrafos de un grafo

Subgrafo expandido

Un subgrafo expandido de un grafo G , es un subgrafo que contiene todos los nodos de G .

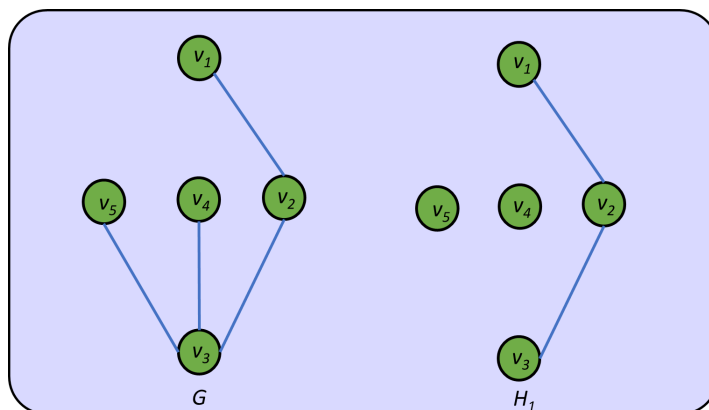


Figura 2.10: Grafo y subgrafo expandido

Subgrafo inducido

Para cualquier subconjunto W de nodos de un grafo G , llamaremos *subgrafo inducido* por W , y lo denotaremos por hW_i , al subgrafo de G que se obtiene tomando los nodos de W y los arcos de G que son incidentes con ellos.



Por tanto, dos nodos de W son adyacentes en hW_i si, y sólo si son adyacentes en G .

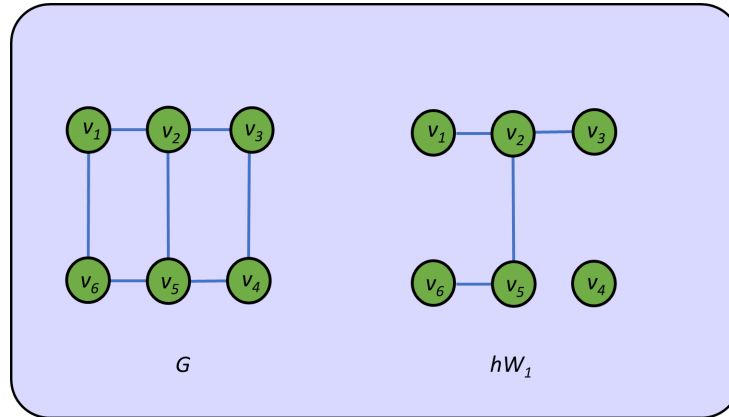


Figura 2.11: Grafo y subgrafo inducido

Subgrafo generador

Dado un grafo $G = (N(G), A(G))$ y $T = (N(T), A(T))$ un subgrafo de G , diremos que T es un subgrafo generador de G si $N(T) = V(T)$ y además que no tiene circuitos.

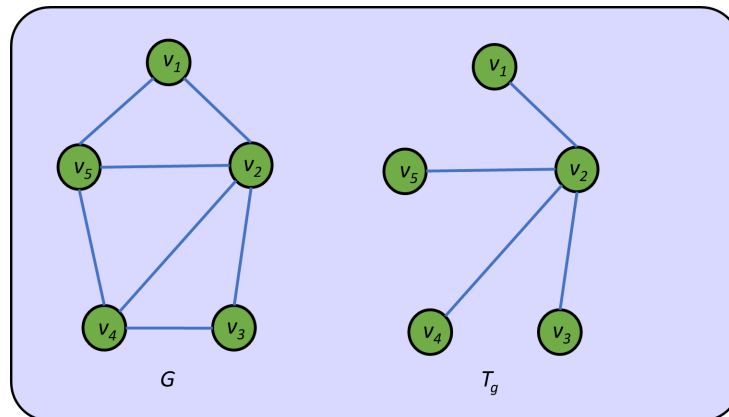


Figura 2.12: Grafo y subgrafo generador

Subgrafo recubridor

Dado un grafo $G = (N, A)$, diremos que un subgrafo $G_1 = (N_1, A_1)$ es un subgrafo recubridor si $N_1 = N$.

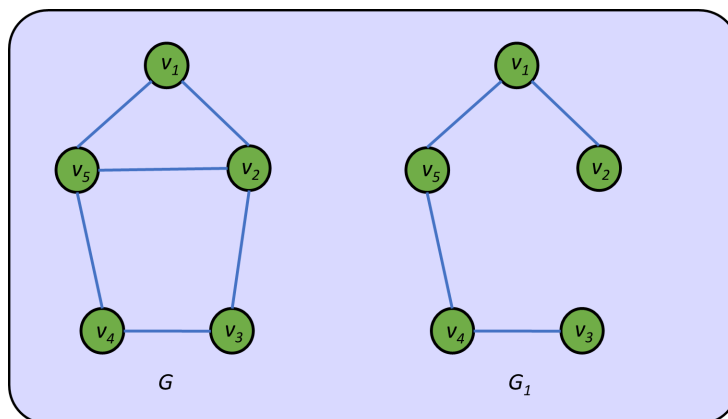


Figura 2.13: Grafo y subgrafo recubridor

2.1.5. Componentes conexos de un grafo

Dado un grafo $G = (N, A)$, las clases de equivalencia definidas en el conjunto de sus nodos, N , por la relación de equivalencia *estar conectado con* reciben el nombre de **componentes conexos de G** .

Observar que de esta forma un grafo no conexo G puede ser *partido* por la relación anterior en subgrafos conexos que son las citadas componentes conexas de G .

2.1.6. Árbol y bosques

Hay un tipo especial de grafo, denominado **árbol**, que se presenta en múltiples aplicaciones.

En ciencias de la computación los arboles son particularmente útiles. Por ejemplo se utilizan para organizar información de tal modo que sea posible efectuar eficientemente operaciones que están relacionadas con esa información.

Para construir algoritmos eficientes para localizar artículos en una lista. Para construir códigos eficientes para almacenar y transmitir datos. Para modelar procedimientos que son llevados a cabo al utilizar una secuencia de decisiones.

Un árbol es un grafo $T = (G)$ acíclico y conexo.

Dado que un árbol no puede tener ciclos, no podría contener ni arcos múltiples ni bucles, por lo tanto cualquier árbol debe ser un grafo simple donde todos sus nodos están unidos por arcos.

A los nodos de un árbol de grado 1 se les denomina *hojas*.



Características de los árboles

Teorema Si $T = (G)$ es un grafo de orden n , ambos son equivalentes

1. T es un árbol
2. Entre cualquier par de nodos de T existe un único camino
3. T es conexo y toda arista de T es una arista de conexión
4. T es acíclico y maximal respecto a esta propiedad
5. T es conexo y tiene tamaño $n - 1$
6. T es acíclico y tiene tamaño $n - 1$

Un problema importante que está asociado a lo representado por un grafo consiste en obtener un árbol que contenga todos los nodos del grafo y algunas de sus arcos para asegurar la conectividad, es decir, un *árbol generador* del grafo.

Árbol Generador

Un árbol generador (*spanning tree*) de un grafo G es un subgrafo generador de G que además es un árbol.

Teorema Todo grafo conexo G contiene un árbol generador.

Hay varias técnicas para obtener un árbol generador correspondiente a un grafo G dado.

Un enfoque consiste en eliminar las arcos que pertenezcan a ciclos, una tras otra hasta que no queden ciclos en el grafo.

Método Cutting-down

- Se empieza por elegir cualquier ciclo de un grafo G conexo y eliminar una de sus arcos (si no hubieran ciclos, el grafo G por sí mismo sería un árbol generador).
- Ya que no se puede desconectar un grafo al eliminar una arista e de un ciclo, el grafo resultante $G - e$ sigue siendo conexo.
- A continuación se repite el proceso hasta que no queden ciclos en el grafo resultante.

Este procedimiento dará lugar al árbol generador T que se está buscando.

Un enfoque alternativo (y posiblemente más eficiente) para la obtención de un árbol generador es la selección de una sucesión de $n - 1$ arcos, una a una, tales que en cada paso el subgrafo actual sea acíclico.



Método Building-up

- Seleccionar los arcos del grafo G de una en una, de tal modo que en cada selección no se cree ningún ciclo, y repetir este procedimiento hasta que todos los nodos del grafo se hayan incluido.

A veces es conveniente considerar a uno de los nodos de un árbol como especial, a dicho nodo se le denomina **raíz**.

Un árbol con una raíz fija es un **árbol enraizado**.

Seleccionar una raíz r en un árbol T requiere de un ordenamiento parcial sobre $V(T)$ en tanto $x \leq y$ si $x \in T_y$.

2.1.7. Diferentes formas de caracterizar el concepto de árbol

Gráficamente existen diferentes formas de representar un árbol y todas ellas son equivalentes, a continuación algunas de ellas:

Desde la teoría de conjuntos

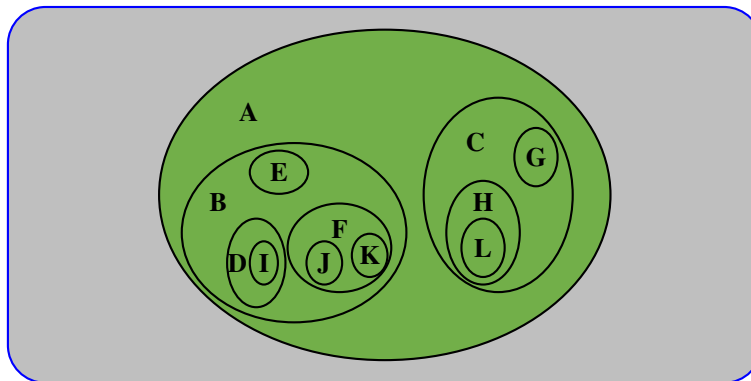


Figura 2.14: Diagrama de Venn

El diagrama anterior se expresa mediante anidación de paréntesis como:

$(A(B(D(I), E, F(J, K)), C(G, H(L))))$



Desde la notación decimal de Dewey

```

1.A,
  1.1.B,
    1.1.1.D,
      1.1.1.1.I,
    1.1.2.E,
    1.1.3.F,
      1.1.3.1.J,
      1.1.3.2.K
  1.2.C,
    1.2.1.G,
    1.2.2.H,
      1.2.2.1.L
  
```

Desde la notación indentada

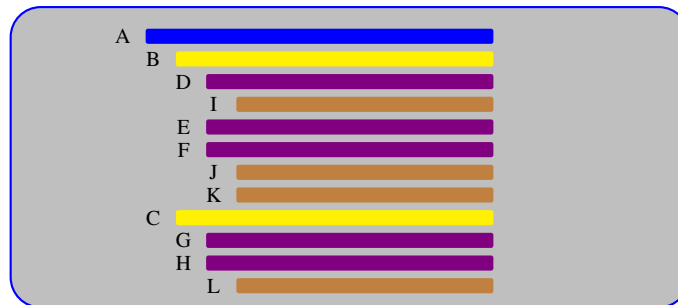


Figura 2.15: Notación indentada

Desde la teoría de grafos

Los árboles se pueden definir como un tipo restringido de **grafo** (fig. 2.16).

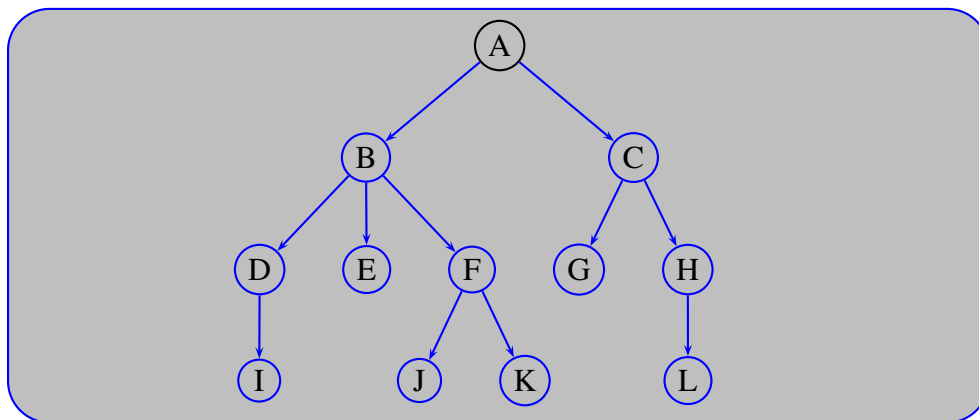


Figura 2.16: Grafo



Un grafo consiste de un número de nodos (puntos o nodos) y un grupo de arcos que unen pares de nodos, a todos los pares de nodos unidos por un arco se les llama nodos adyacentes y los arcos pueden tener una dirección determinada, generando así un grafo dirigido (de lo contrario sería no-dirigido).

Por convención a los nodos de un grafo se les representa con círculos y los arcos que los conectan como líneas (no-dirigido) o flechas (dirigido).

Esta representación es la que más comúnmente se ha utilizado y ha originado el término árbol por su parecido con el vegetal (raíz, ramas, hojas).

Los árboles son un subconjunto importante de los grafos, y una herramienta útil para describir estructuras que representan algún tipo de jerarquía.



2.2. Guardando un grafo en una matriz de adyacencias

Resolver problemas de usando la representación de grafos



n general, el proceso de resolver un problema utilizando grafos se puede resumir de la siguiente manera:

1. Identificar el tipo de problema y el algoritmo que se aplicará
2. Escoger la representación del grafo (normalmente depende del algoritmo)
3. Decidir cómo etiquetar los nodos del grafo
4. Leer la entrada y construir la representación del grafo
5. Aplicar el algoritmo elegido
6. Imprimir el resultado en el formato que se pide

Como ya se mencionó es posible representar computacionalmente un grafo G mediante una matriz de adyacencias.

Pero esta no es la única forma de hacerlo, a continuación se enlistan algunas formas de hacerlo:

1. **Matriz de adyacencias:** los arcos del grafo se guardan en una matriz, indicando si un nodo tiene enlace directo con otro
2. **Lista de adyacencia:** para cada nodo, sólo se guardan sus vecinos
3. **Lista de arcos:** se guarda una lista de todos los arcos del grafo

Dependerá de la naturaleza del problema la forma en la que se represente la estructura, en el caso de matrices de adyacencia existen algunos casos en los que el llenado de la matriz puede hacerse en forma automática, a continuación algunos de ellos.

Llenado automático

Ejemplo 1

Se define un grafo con 5 nodos. Se inicializa la matriz con 0's y se iguala a 1 la diagonal principal.



```

v1 v2 v3 v4 v5
v1 1 0 0 0 0
v2 0 1 0 0 0
v3 0 0 1 0 0
v4 0 0 0 1 0
v5 0 0 0 0 1

Process exited with return value 0
Press any key to continue . . .

```

Figura 2.17: Desplegado de la matriz resultante

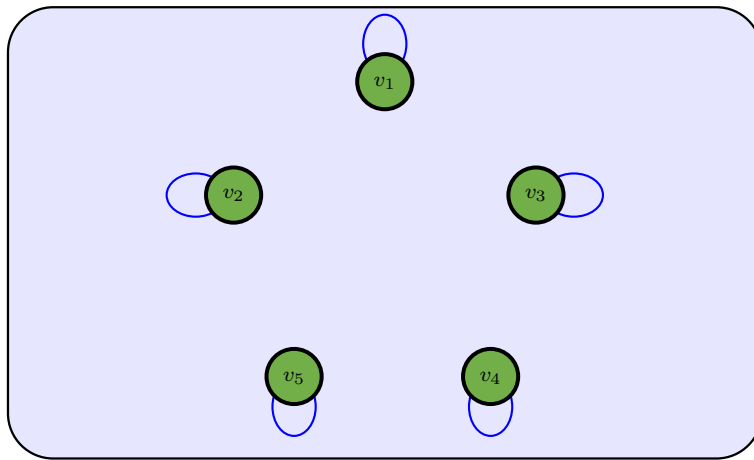


Figura 2.18: Grafo resultante

Y el grafo producido se muestra a continuación:

Como puede apreciarse, el grafo producido es cíclico no conexo.

Ejemplo 2

Se define un grafo con 5 nodos. Se inicializa la matriz con 1's y se iguala a cero la diagonal principal.

```

v1 v2 v3 v4 v5
v1 0 1 1 1 1
v2 1 0 1 1 1
v3 1 1 0 1 1
v4 1 1 1 0 1
v5 1 1 1 1 0

Process exited with return value 0
Press any key to continue . . .

```

Figura 2.19: Desplegado de la matriz resultante

Como puede apreciarse, el grafo producido es cíclico conexo completo.

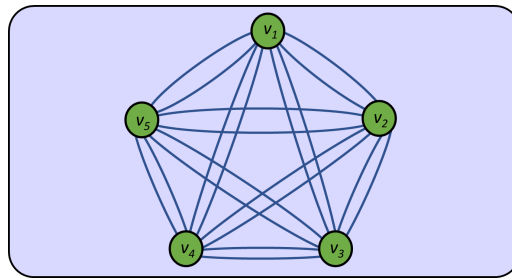


Figura 2.20: Grafo resultante

Ejemplo 3

Se define un grafo con 5 nodos. Se inicializa la matriz con 0's y se iguala a 1 la mitad superior encima de la diagonal principal.

```

v1 v2 v3 v4 v5 f...
v1 0 1 1 1 1
v2 0 0 1 1 1
v3 0 0 0 1 1
v4 0 0 0 0 1
v5 0 0 0 0 0
.....
Process exited with return value 0
Press any key to continue . . .

```

Figura 2.21: Despliegado de la matriz resultante

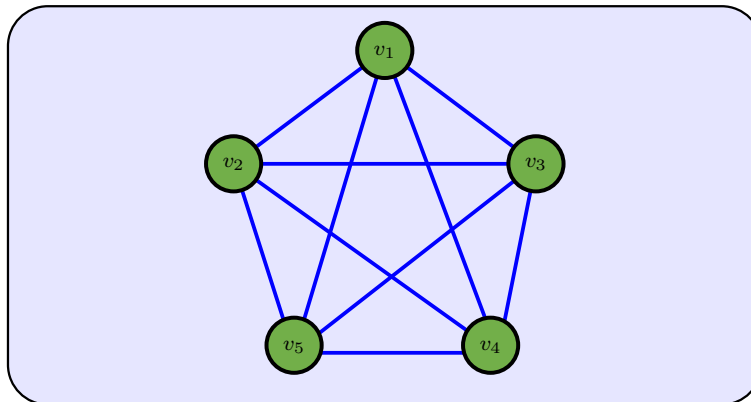


Figura 2.22: Grafo resultante

Como puede apreciarse, el grafo producido es conexo completo.



2.3. ¿Cómo hacer un algoritmo para visitar cada nodo de un grafo almacenado en la computadora?

Recorridos de un grafo



En una estructura lineal resulta trivial establecer un criterio de movimiento por la misma para acceder a los elementos, pero en un árbol esa tarea no resulta tan simple.

Los árboles tienen una gran variedad de aplicaciones.

Para construir un árbol genealógico, para el análisis de circuitos eléctricos y para numerar los capítulos y secciones de un libro, en el área de computación: diseño de compiladores, sistemas expertos, sistemas evolutivos, sistemas conscientes, manejo de directorios, índices de bases de datos y muchas más.

No obstante, existen distintos métodos útiles en que podemos sistemáticamente recorrer todos los nodos de un árbol.

Algoritmo primero en profundidad y algoritmo primero en anchura

Hay dos formas básicas de recorrer un árbol:

- El recorrido en amplitud
- El recorrido en profundidad

La forma de recorrido más sencilla es el recorrido en amplitud, el cual se realiza avanzando por niveles desde la raíz hasta las hojas avanzando de izquierda a derecha.

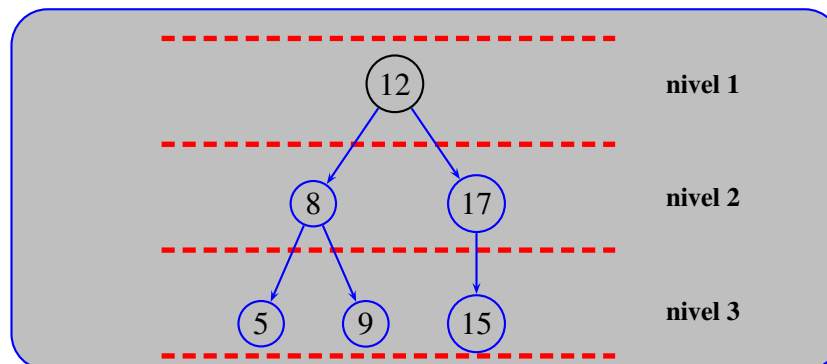


Figura 2.23: $Recorrido_{amplitud} = \{12, 8, 17, 5, 9, 15\}$



En el recorrido en **profundidad** se recorre el árbol por subárboles y existen tres formas, cada una de ellas tiene una secuencia distinta para analizar el árbol, éstas se denominan **preorden**, **inorden** y **postorden**.

El recorrido en **preorden** (fig.2.24) es:

- Si el árbol tiene un único elemento, dicho elemento es el listado en preorden
- Si el árbol tiene más de un elemento, el listado en preorden es listar el nodo raíz seguido del listado en preorden de cada uno de los subárboles hijos de izquierda a derecha

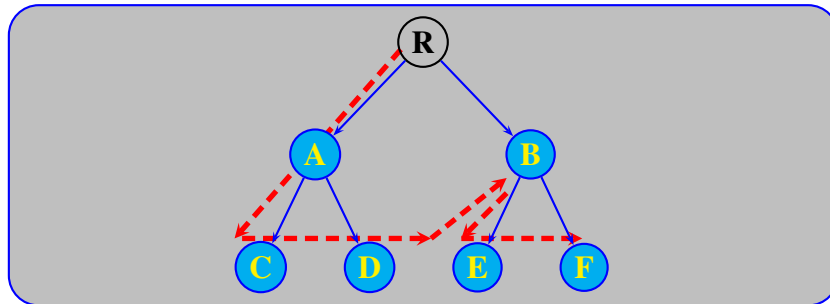


Figura 2.24: $Recorrido_{preorden} = \{R, A, C, D, B, E, F\}$

El recorrido en **inorden** (fig.2.25) es:

- Si el árbol tiene un único elemento, dicho elemento es el listado en inorden
- Si el árbol tiene más de un elemento, el listado en inorden es listar el subárbol A1 en inorden, y listar el nodo raíz seguido del listado en inorden de cada uno de los subárboles hijos de izquierda a derecha restantes

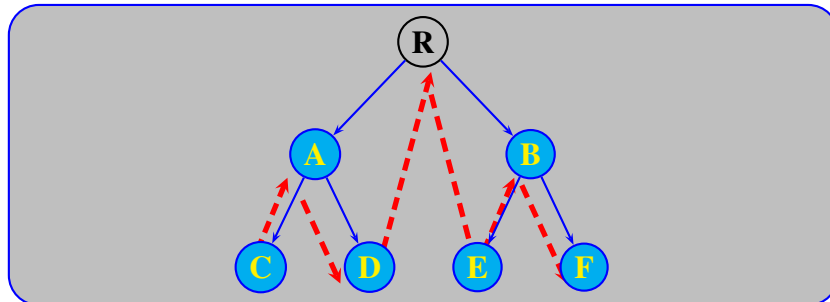


Figura 2.25: $Recorrido_{inorden} = \{C, A, D, R, E, B, F\}$

El recorrido en **postorden** (fig.2.27) es:



- Si el árbol tiene un único elemento, dicho elemento es el listado en postorden
- Si el árbol tiene más de un elemento, el listado en postorden es listar en postorden cada uno de los subárboles hijos de izquierda a derecha seguidos por el nodo raíz

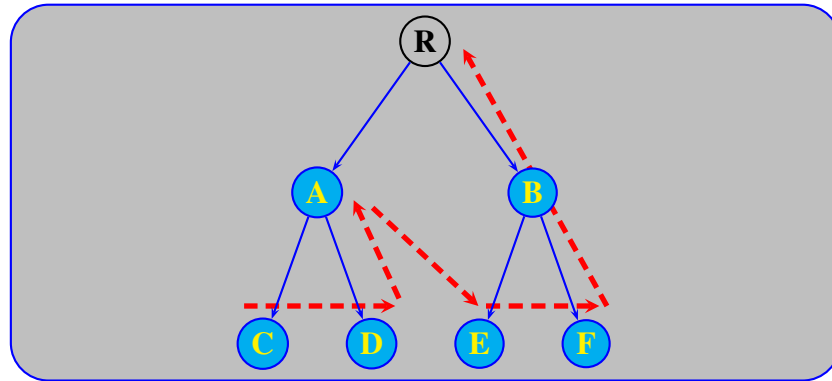


Figura 2.26: $\text{Recorrido}_{\text{postorden}} = \{C, D, A, E, F, B, R\}$

A continuación el conjunto de operaciones primitivas consideradas en los árboles:

- **CREAR_RAIZ(u)**: Construye un nuevo nodo r con etiqueta u y sin hijos, devuelve el árbol con raíz r , es decir, un árbol con un único nodo.
- **DESTRUIR(T)**: Libera los recursos que mantienen el árbol T de forma que para volver a usarlo se debe de asignar un nuevo valor con la operación de creación.
- **PADRE(n, T)**: Esta función devuelve el padre del nodo n en el árbol T , si n es la raíz, que no tiene padre, devuelve NODO_NULO (un valor que será usado para indicar que se ha intentado salir del árbol). Como precondition n no es NODO_NULO (por tanto T no es vacío).
- **HIJO_IZQDA(n, T)**: Devuelve el descendiente más a la izquierda en el siguiente nivel del nodo n en el árbol T , y devuelve NODO_NULO si n no tiene hijo a la izquierda. Como precondition n no es NODO_NULO.
- **HERMANO_DRCHA(n, T)**: Devuelve el descendiente a la derecha del nodo n en el árbol T , definido para ser aquel nodo m con el mismo padre que n , es decir, padre p , de tal manera que m cae inmediatamente a la derecha de n en la ordenación de los hijos de p . Devuelve NODO_NULO si n no tiene hermano a la derecha. Como precondition n no es NODO_NULO.



- **ETIQUETA(n, T):** Devuelve la etiqueta del nodo n en el árbol T (se manejarán árboles etiquetados, sin embargo no es obligatorio definir etiquetas para cada árbol). Como precondition n no es NODO_NULO.
- **REETIQUETA(e, n, T):** Asigna una nueva etiqueta e al nodo n en el árbol T . Como precondition n no es NODO_NULO.
- **RAIZ(T):** Devuelve el nodo que está en la raíz del árbol T o NODO_NULO si T es el árbol vacío.
- **INSERTAR_HIJO_IZQDA(n, T_i, T):** Inserta el árbol T_i como hijo a la izquierda del nodo n que pertenece al árbol T . Como precondition n no es NODO_NULO y T_i no es el árbol vacío.
- **INSERTAR_HERMANO_DRCHA(n, T_d, T):** Inserta el árbol T_d como hermano a la derecha del nodo n que pertenece al árbol T . Como precondition n no es NODO_NULO y T_d no es el árbol vacío.
- **BORRAR_HIJO_IZQDA(n, T):** Devuelve el subárbol con raíz hijo a la izquierda de n del árbol T el cual se ve privado de éstos nodos. Como precondition n no es NODO_NULO.
- **BORRAR_HERMANO_DRCHA(n, T):** Devuelve el subárbol con raíz hermano a la derecha de n del árbol T el cual se ve privado de estos nodos. Como precondition n no es NODO_NULO.

2.3.1. Árboles Generales

Cuando cualquier nodo del árbol puede tener un número arbitrario de nodos hijos, a esto se le conoce como un árbol general.

Un árbol general con nodos de tipo **T** es un par (r, L_A) , formado por un nodo r (la raíz) y una lista L_A de árboles generales del mismo tipo.

Esta lista se conoce como **bosque** y puede estar vacía. Cada uno de los árboles del bosque es un hijo. Si el orden de los hijos no es relevante L_A , es un conjunto en vez de una lista.

Es decir, con esta definición, no puede haber árboles generales vacíos, sino que tiene que haber al menos un elemento, la raíz.

En un árbol general se define el grado de un nodo como el número de hijos de ese nodo y el grado del árbol como el máximo de los grados de los nodos del árbol.

La representación de un árbol general plantea algunas dificultades:

- ¿Puede que el número de hijos por nodo varíe en un rango grande (posiblemente sin límite lógico superior)?



- Establecer un límite superior hace que algunos árboles no se puedan representar. Incluso con un límite superior, la asignación de un número fijo de apuntadores en cada nodo puede perder una gran cantidad de espacio.
- ¿Cómo pueden organizarse los apuntadores para un fácil recorrido?
- ¿Requiere esto una estructura de datos secundaria dentro del nodo?
- ¿Tal esquema proporciona una búsqueda eficiente?

2.3.2. Características y propiedades de los árboles

Otra característica que normalmente tiene los árboles es que todos los nodos contengan el mismo número de apuntadores, es decir, se usa la misma estructura para todos los nodos del árbol.

Esto hace que la estructura sea más sencilla, y por lo tanto también los programas para trabajar con ellos. Tampoco es necesario que todos los nodos hijos de un nodo concreto existan. Es decir, que pueden usarse todos, algunos o ninguno de los apuntadores de cada nodo.

Un árbol en el que todos o ninguno de los hijos existe, se llama árbol completo. Los árboles se parecen al resto de las estructuras que se han visto en que, dado un nodo cualquiera de la estructura, este se puede considerar como una estructura independiente. Es decir, un nodo cualquiera puede ser considerado como la raíz de un árbol completo.

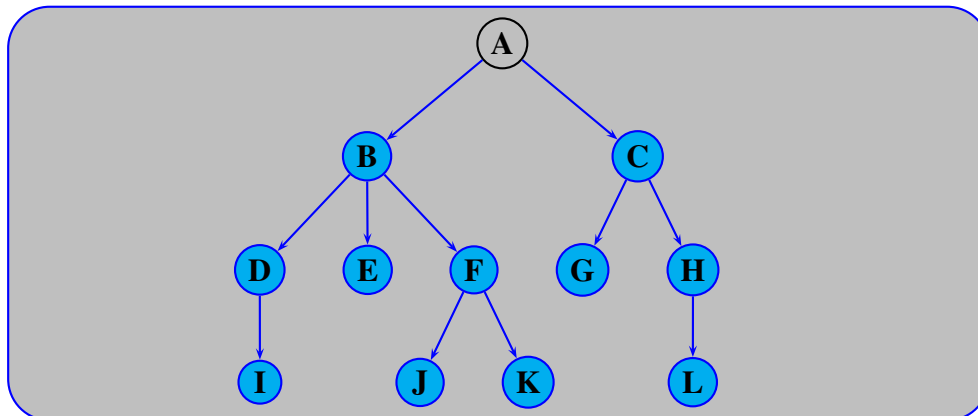


Figura 2.27: Árbol general

Dado el árbol general de la figura 2.27, se hacen sobre él tenemos lo siguiente:

1. A es la raíz del árbol.
2. B es hijo de A. C es hijo de A. D es hijo de B. E es hijo de B. L es hijo de H.



3. A es padre de B. B es padre de D. D es padre de I. C es padre de G. H es padre de L.
4. B y C son hermanos. D,E y F son hermanos. G y H son hermanos. J y K son hermanos.
5. I, E, J, K, G y L son nodos terminales u hojas.
6. B, D, F, C y H son nodos interiores.
7. El grado del nodo A es 2. B es 3. C es 2. D es 1. E es 0. El grado del árbol es 3.
8. El nivel del nodo A es 1. B es 2. D es 3. C es 2. L es 4.
9. La altura del árbol es 4.

2.3.3. Longitud de camino interno y externo

Longitud de camino (LC)

Se define la longitud de camino X como el número de arcos que deben ser recorridos para llegar desde la raíz al nodo X. por definición la raíz tiene longitud de camino 1, sus descendientes directos 2...

Por ejemplo el nodo I del árbol de la figura 2.27, tiene una LC igual a 4.

Longitud de camino interno (LCI)

Es la suma de las longitudes de camino de todos los nodos del árbol, y se calcula por medio de la fórmula 2.1.

$$LCI = \sum_{i=1}^h n_i * i \quad (2.1)$$

Donde:

i = nivel del árbol

h = altura del árbol

n_i = número de nodos en el nivel i

La media de la LCI se calcula mediante la ecuación 2.2:

$$LCI_M = LCI/N \quad (2.2)$$

Donde:

N = número de nodos

**Longitud de camino externo (LCE)**

Es la suma de las longitudes de camino de todos los nodos especiales del árbol y se calcula con la formula:

$$LCE = \sum_{i=2}^{h+1} ne_i * i \quad (2.3)$$

Donde:

i = nivel del árbol

h = altura del árbol

ne = nodo especial ²

²**Nodos especiales:** Tienen como objetivo remplazar las ramas vacías o nulas, no pueden tener descendientes y normalmente se representa en forma de cuadrado.



2.4. Algoritmo para calcular todas las componentes conexas de un grafo

Analizando las potencias de la matriz de adyacencia puede estudiarse la conexión de un grafo de manera eficaz, pero poco eficiente desde el punto de vista del cómputo operacional.

Yendo a la definición de conexión (existencia de caminos entre cada par de nodos) y construyendo nuevos caminos a partir de los ya existentes se obtienen mejores algoritmos para probarlo.

El mas sencillo, y que sirve como base a otros interesantes algoritmos es el algoritmo de Warsall [Warshall, 1962].

Algoritmo de Warsall

Este algoritmo recrea la construcción de trayectorias entre nodos de la manera siguiente:

Los nodos v_i y v_j están conectados si hay un camino entre ellos o, si para algún nodo v_k hay un camino de v_i a v_k y un camino de v_k a v_j .

La estrategia que sigue el algoritmo es de comprobaciones exhaustivas y lo hace en sentido contrario al sugerido por la frase anterior: no comprueba si para cada par de nodos v_i, v_j hay algún otro v_k que hace de enlace, sino al revés, comprueba si cada nodo v_k es puente entre cada par de nodos v_i y v_j .

Warsall produce una sucesión de matrices booleanas W_1, W_2, \dots, W_n (una por cada nodo v_k a comprobar) que indican si dos nodos dados están o no conectados.

Algoritmo 1.- (de Warsall)

```

inicio: n; M;  $W_0 = M$ 
para k = 1 hasta n
    para j = 1 hasta n
        para i = 1 hasta n
             $W_k(i, j) = W_{k-1}(i, j) \mid (W_{k-1}(i, k) \& W_{k-1}(k, j))$ 
        fin
    fin
fin

```

Figura 2.28: Seudo código Warsall

Resultado

Si el grafo es conexo la última matriz constará toda de unos (si hay al menos dos nodos). Inicialmente, comenzaremos usando la matriz de adyacencia M como matriz de conexión inicial ($W_0 = M$, sólo están conectados los nodos extremos de los arcos).



En el paso 1, y para cada v_i y v_j , se comprueba si ya están conectados o si pueden conectarse a través de v_1 (es decir, si v_i está conectado con v_1 y también v_1 está conectado con v_j).

Obtendremos así una nueva matriz W_1 que indicará los nodos conectados, bien porque lo estaban o bien porque se han conectado a través de v_1 . Y se repite lo mismo para cada uno de los nodos restantes, hasta obtener lo que se llama **matriz de clausura transitiva**.



2.5. Algoritmo que encuentra el camino más corto, entre dos nodos de un grafo

Quando un grafo representa un modelo específico, sus arcos tienen características propias, que pueden representar longitudes físicas, costos monetarios, tiempos necesarios para recorrerlas, por lo que en un grafo se va asignando a cada arista la magnitud que se desea considerar (es decir, su *peso*).

Así, cuando a cada arista de un grafo se le ha asignado un valor si se busca el *camino más corto* entre dos nodos dados, no se considera que es el que tiene menor número de arcos, sino el que recorre en total la menor distancia.

Se denomina **peso** de un grafo $G = (V, A)$ a aquella función real positiva sobre el conjunto de arcos del grafo dada por:

$$\omega : A \rightarrow R^+$$

Donde el peso de cada arista se va a denotar como $\omega(\{v_i, v_j\}) = \omega_{ij}$, y se va a llamar **peso de una trayectoria** a la suma de los pesos de las arcos que la componen.

En un grafo pesado, se llama **matriz de pesos del grafo** a la matriz $\Omega = (\omega_{ij})_{n \times n}$, donde se asigna $\omega_{ij} = \infty$ si no hay arista desde el nodo v_i al nodo v_j y $\omega_{ii} = 0$, ceros en la diagonal.

Se va a denominar *peso mínimo* de v_i a v_j al mínimo de los pesos de los caminos de v_i a v_j , representado por ω_{ij}^* , y *camino* (de peso) *mínimo* de v_i a v_j a cualquier camino C_{ij} entre ellos que tenga peso mínimo (es decir, $\omega(C_{ij}) = \omega_{ij}^*$).

La *matriz de pesos mínimos* es la matriz $\Omega^* = (\omega_{ij}^*)_{n \times n}$ con $\omega_{ij}^* = \infty$ si no hay camino de v_i a v_j .

Proposición Sea G un grafo pesado. Si $x_1 x_2 \dots x_{p-1} x_p$ es un camino mínimo, para cada i con $1 < i < p$, los caminos $x_1 x_2 \dots x_i$ y $x_i \dots x_{p-1} x_p$ son caminos mínimos y $\omega_{1p}^* = \omega_{1i}^* + \omega_{ip}^*$.

Algoritmo de Dijkstra

Uno de los algoritmos más usados para la búsqueda de caminos de peso mínimo es el de Dijkstra [Dijkstra, 1959], que proporciona los pesos mínimos desde un nodo dado al resto de los nodos.

El algoritmo devuelve en realidad el peso mínimo no el camino mínimo propiamente dicho, pero permite obtener fácilmente el camino mínimo recorriendo en sentido inverso la construcción (de ahí su popularidad).

Sea un grafo pesado, con $V = \{v_1, v_2, \dots, v_n\}$ su conjunto de nodos y su matriz de pesos $\Omega = (\omega_{ij})_{n \times n}$, y sea v_p el nodo inicial.



Dijkstra construye, en cada paso, un camino mínimo desde v_p a otro nodo y se detiene cuando ha construido uno para cada nodo (o no puede construir más). Para ello se usan una lista o conjunto: L , que contendrá los nodos para los que ya hemos construido un camino mínimo y un vector de pesos: D , que contendrá al final los pesos mínimos. Inicialmente $L = \{v_p\}$ y $D = \Omega(p, :)$, la p -ésima fila de la matriz de pesos (la correspondiente al nodo inicial).

Algoritmo 4.- (de Dijkstra)

```

inicio:  $\Omega$ ;  $v_p$ ;  $L = \{v_p\}$ ;  $D = \Omega(p, :)$ 
mientras sea  $V - L \neq \emptyset$ 
    tomar  $v_k \in V - L$  con  $D(k)$  mínimo
    hacer  $L = L \cup \{v_k\}$ 
    para cada  $v_j$  de  $V - L$ 
        si  $D(j) > D(k) + \omega_{kj}$ 
            hacer  $D(j) = D(k) + \omega_{kj}$ 
        fin
    fin
fin

```

Figura 2.29: Algoritmo de Dijkstra

El vector D final contiene los pesos mínimos desde el nodo inicial a los demás nodos -si alguno de los pesos finales es ∞ , no hay camino desde el nodo inicial-. El algoritmo de Dijkstra es un algoritmo voraz (*greedy*) que genera uno a uno los caminos de un nodo a al resto por orden creciente de longitud; usa un conjunto S de nodos donde, a cada paso del algoritmo, se guardan los nodos para los que ya se sabe el camino mínimo y devuelve un vector indexado por nodos, de modo que para cada uno de estos nodos se puede determinar el costo de un camino más económico (de peso mínimo) de a a tales nodos.

Cada vez que se incorpora un nodo a la solución, se comprueba si los caminos todavía no definitivos se pueden acortar pasando por él.



2.6. ¿Cómo hacer un algoritmo para saber si un grafo almacenado en la computadora es un árbol o no?

Un árbol es un grafo no dirigido acíclico, conectado sin circuitos simples.

Un grafo no dirigido es un árbol si y sólo si existe una ruta única simple entre cualquiera dos de sus nodos.

Sea $G = (V, A)$ un árbol. Entonces:

- Entre cada par de nodos u, v hay un único camino.
- Al quitar de A cualquier arista resulta un bosque con 2 árboles.
- Al añadir una arista nueva siempre se obtiene un ciclo $|A| = |V| - 1$.

El algoritmo que indica sí un grafo es o no un árbol deberá considerar las siguientes condiciones:

Sí:

- La diagonal principal de la matriz de adyacencia es cero, lo que demuestra que no existan ciclos.
- Y, que al menos exista un 1 fuera de la diagonal principal, lo que demuestra que hay al menos un par de nodos conectados.
- Que solamente exista un 1 (conexión) por fila y por columna, lo que demuestra que no hay circuitos.



2.7. Árbol de expansión

2.7.1. Algoritmos para encontrar el árbol de expansión mínima o máxima



rocede del grafo de expansión (2.1.4) donde, dado un grafo conexo, no dirigido G , un **árbol de expansión** es un árbol compuesto por todos los nodos y algunas (posiblemente todas) de las arcos de G .

Al ser creado un árbol no existirán ciclos, además debe existir una ruta entre cada par de nodos. Un grafo puede tener muchos árboles de expansión.

Árbol Minimal o Maximal

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

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

Árbol de Expansión Mínima

Dado un grafo conexo, no dirigido y con pesos en las arcos, un árbol de expansión mínima es un árbol compuesto por todos los nodos y cuya suma de sus arcos es la de menor peso.

El problema de hallar el *Árbol de Expansión Mínima* (**MST**³) puede ser resuelto con varios algoritmos, los mas conocidos son los de Kruskal [Kruskal, 1956] y Prim [Prim, 1957].

2.7.2. Algoritmo de Kruskal

Es un algoritmo voraz que tiene como entrada un grafo conexo, pesado y como salida un árbol *minimal* o *maximal*, donde el peso total de las arcos en el árbol es maximizada o minimizada según corresponda.

Como funciona:

Primero se ordenan las arcos del grafo por su peso de menor a mayor. Mediante la técnica greedy Kruskal se intentará unir cada arista siempre y cuando no se forme un ciclo.

³MST: proviene del inglés *Minimum Spanning Tree*.



Como se han ordenado los arcos por peso se comienza con la arista de menor peso, si los nodos que contienen dicha arista no están en la misma componente conexa entonces se unen para formar una sola componente, se revisa si están o no en la misma componente conexa al hacer esto se evita que se creen ciclos y que la arista que une dos nodos siempre sea la mínima posible.

Algoritmo 1: Algoritmo de Kruskal

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

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

```

1  INICIO
2  | Ordenar los pesos de los nodos del Grafo  $G$  de menor a mayor
3  |  $n \leftarrow$  el número de nodos de  $G$ 
4  |  $T \leftarrow \emptyset$ 
5  | Inicializar  $n$  conjuntos, que contengan un elemento diferente de  $V$ 
6  | repeat
7  |    $e \leftarrow \{u, v\};$            // el menor arco aun no considerado
8  |    $ucomp;$                      // hallar (u)
9  |    $vcomp;$                      // hallar (v)
10 |   if  $ucomp \neq vcomp$  then unir( $ucomp, vcomp$ );
11 |    $T \leftarrow T \cup e$  endif
12 | until  $T$  contenga  $(n - 1)$  arcos ;
13 FIN
  
```

Sí a es el número de arcos, la complejidad del algoritmo de Kruskal es $O(a \log n)$.

2.7.3. Algoritmo de Prim

Es un algoritmo voraz que tiene como entrada un grafo conexo, pesado y como salida un árbol *minimal* o *maximal*, donde el peso total de las arcos en el árbol es maximizada o minimizada según corresponda.

Como funciona:

El algoritmo incrementa continuamente el tamaño de un árbol, comenzando por un nodo inicial al que se le van agregando sucesivamente nodos cuya distancia a los anteriores es mínima. Esto significa que en cada paso, las arcos a considerar son aquellas que inciden en nodos que ya pertenecen al árbol.

El árbol de expansión mínimo está completamente construido cuando no quedan más nodos por agregar.



Algoritmo 2: Algoritmo de Prim

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

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

1 **INICIO**

2 $T \leftarrow \emptyset$

3 $B \leftarrow \{\text{Un nodo arbitrario de } V\}$

4 **while** $B \neq V$ **do**

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

6 $T \leftarrow T \cup \{e\};$

7 $B \leftarrow B \cup \{v\}$

8 **endw**

9 **FIN**

10

Donde:

- B : conjunto solución.

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 $O(n^2)$.

Comparación general

Tenemos las siguientes diferencias y similitudes entre los algoritmos de Kruskal y Prim:

- Ambos algoritmos son del tipo voraz (*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 nodo 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.

Con base en su complejidad un criterio para determinar cuál algoritmo usar en un caso específico sería el siguiente:

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

2.7.4. Aplicaciones reales donde encontrar el árbol de expansión mínima y máxima es la solución del problema

Ver sección de ejercicios.


Unidad 3

GRAFOS BIPARTITOS, PROPIEDADES, ALGORITMOS Y APLICACIONES

“Comprender e interpretar el concepto de grafo bipartito y sus propiedades, implementar y aplicar algoritmos para grafos bipartitos, conocer e identificar aplicaciones de la modelación de problemas con grafos bipartitos”

Objetivos específicos 1, 2 y 3

3.1. Introducción

 Los grafos vistos hasta ahora poseen nodos que tienen la misma naturaleza (o clase): computadoras en una red, átomos en una molécula, componentes en un circuito eléctrico (fig.3.1) se consideran elementos de la *misma clase*.

Pero existe otro tipo de grafos en los que se puede partir su conjunto de nodos en dos clases (es decir, sus nodos son la unión de dos grupos de nodos).

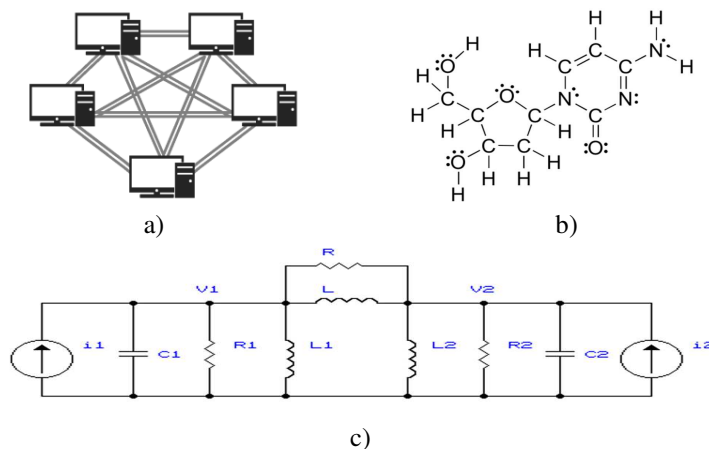


Figura 3.1: Grafos de la misma *clase*, a) nodos = computadoras, b) nodos = elementos químicos, c) nodos = dispositivos eléctricos.

Un ejemplo de grafos bipartitos son los grafos coloreados con dos colores (fig.3.4) donde cada color (en este caso, rojo y azul) se considera como una *clase diferente*.

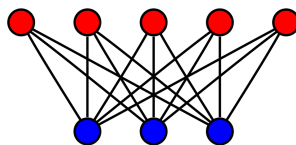


Figura 3.2: Grafo bipartito coloreado

Grafo bipartito completo

El grafo bipartito completo con m y n nodos, denotado $(K_{m,n})$, es un grafo simple cuyo conjunto de nodos está dividido en conjuntos N_1 con m nodos y N_2 con n nodos, de los cuales existe un arco entre cada par de nodos n_1 y n_2 , donde n_1 está en N_1 y n_2 está en N_2 . El grafo de la fig.3.3 es bipartito completo con dos y cuatro nodos ($K_{2,4}$) donde $N_1 = \{3, 5\}$ y $N_2 = \{1, 2, 4, 6\}$

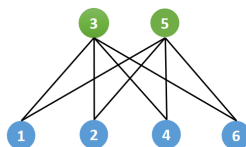


Figura 3.3: Grafo bipartito completo $K_{2,4}$



3.2. Concepto de grafo bipartito

Definición (► Grafo bipartito)

Un grafo $G = (N, A)$ es **bipartito** si existe una partición del conjunto de nodos, es decir si $N = N_1 \cup N_2$, con $N_1 \cap N_2 = \emptyset$, de manera que los arcos existentes solo conectan nodos de N_1 con nodos de N_2 : es decir, $\{u, v\} \in A$ implica que $u \in N_1, v \in N_2$ o $v \in N_1, u \in N_2$. De manera equivalente, si $\{u, v\} \in A, u \in N_1 \Leftrightarrow v \in N_2$.

Los grafos bipartitos se representan gráficamente con dos columnas (o filas, fig.3.4) de nodos y los arcos uniendo nodos de columnas (o filas) diferentes.

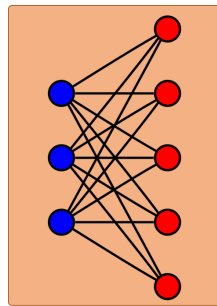


Figura 3.4: Grafo bipartito en dos columnas

Los dos conjuntos U y N pueden ser definidos como un coloreo del grafo con dos colores: si se colorean los nodos en U de azul y los nodos de N de rojo se obtiene un grafo de dos colores donde cada arco tiene un nodo azul y el otro rojo. Por otro lado, si un grafo no tiene la propiedad de que se puede colorear con dos colores no es bipartito.

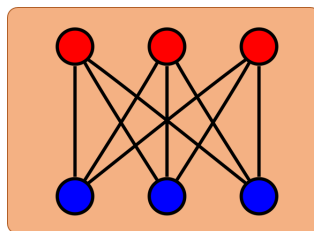


Figura 3.5: Grafo balanceado



Un grafo bipartito con la partición de los nodos en U y N (fig.3.5) suele denotarse $G = (U, N, E)$. Si $|U| = |N|$, esto es, si los dos subconjuntos tienen la misma cantidad de elementos o cardinalidad, decimos que el grafo bipartito G es *balanceado*.

Si todos los nodos del mismo lado de la bipartición tienen el mismo grado, entonces G es llamado grafo *birregular*.

Propiedades

Todo grafo sin ciclos con cantidad de nodos impar es bipartito

- Como consecuencia de esto: Todo árbol es bipartito
 - Los grafos cíclicos con un número par de nodos son bipartitos.
 - Todo grafo planar donde todas las caras tienen un número par de arcos es bipartito.

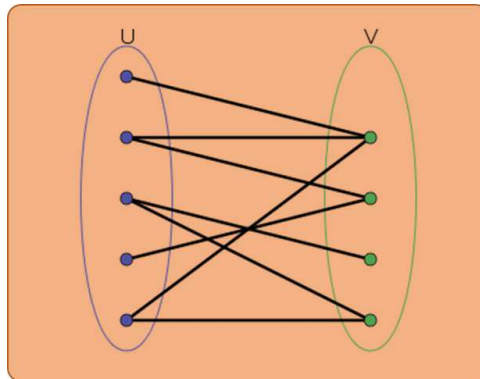


Figura 3.6: Grafo cíclico

Definición (► Grafo bipartito completo)

Un grafo bipartito completo $G = (N_1 \cup N_2, A)$ es un grafo bipartito tal que $\forall v_1 \in N_1, \forall v_2 \in N_2 \Rightarrow a(v_1, v_2) \in A$. Es decir, un grafo bipartito completo está formado por dos conjuntos disjuntos de nodos y todas los arcos que unen esos nodos. El grafo completo bipartito con particiones de tamaño $|N_1| = m$ y $|N_2| = n$, es denotado como $K_{m,n}$.

**Propiedades**

Sea $K_{m,n}$ un grafo bipartito con $|N_1| = m$ y $|N_2| = n$, se verifica:

- $|A| = |N_1||N_2| = mn$
- $K_{m,n}$ posee $m^{n-1}n^{m-1}$ árboles de expansión



3.3. Diferentes formas de caracterizar un grafo bipartito

Hipergrafo

Una representación alternativa de los grafos bipartitos son los **hipergrafos**. Sea $G(N_1, N_2, A)$ un grafo bipartito, es posible construir un hipergrafo derivado de éste $Hyp(G) = (N_1, F)$. Se debe señalar que la noción de hipergrafo amplía la noción de grafo al permitir que los bordes del hipergrafo conecten cualquier número de nodos mediante los llamados *hiperarcos*.

Configurando:

$$F = (\{v \in N_2 | \{u, v\} \in A\})_{u \in N_1}$$

llegamos a un hipergrafo H en el que los nodos izquierdos de G son los nodos del hipergrafo, y los vecinos de cada nodo izquierdo u en N_1 hacen una hiperarco en H . Al contrario, cada hipergrafo se puede reducir a un grafo bipartito de esta manera (otra razón por la cual los gráficos bipartitos son importantes).

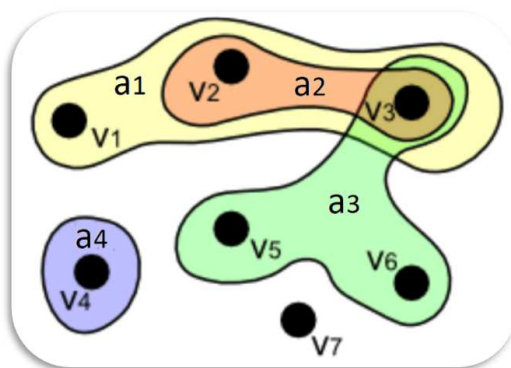


Figura 3.7: Hipergrafo

Por ejemplo, un conjunto de grupos de personas en un sitio de redes sociales pueden ser modeladas como un hipergrafo, en el que las personas son los nodos, y cada grupo crea una hiperarco que está formada por todas las personas que figuran en ese grupo.

El grafo bipartito equivalente tiene personas y grupos como nodos, y un arco de una persona a un grupo cuando la persona es miembro de ese grupo.

Debido a la simetría entre N_1 y N_2 , se puede construir una representación hipergráfica análoga $Hyp(G) = (N_2, F)$ en la que F contiene una hiperarco para cada nodo en N_1 .



Modelo de espacio vectorial

Otra forma alternativa de modelar un grafo bipartito es usar el **modelo de espacio vectorial**. Este enfoque es muy común en la minería de textos¹, donde los documentos que contienen palabras se modelan como vectores de palabras.

Sea $G = (N_1, N_2, A)$ una red de palabras de un documento bipartito, donde N_1 son los documentos y N_2 son las palabras. El modelo de espacio vectorial representa cada documento $u \in N_1$ como un vector.

En el área de recuperación de información normalmente se usa una expresión vectorial, donde las dimensiones del vector representan términos, frases o conceptos que aparecen en el documento.

En este aspecto la representación más adoptada es la conocida como bolsa de palabras: una colección de documentos compuesta por n documentos indexados y m^2 términos representados por una matriz documento-término de $n \times m$.

Donde los n vectores renglón representan los n documentos; y el valor asignado a cada componente refleja la importancia o frecuencia ponderada que produce el término, frase o concepto t_i en la representación semántica del documento j .

En esta representación, ciertas medidas surgen naturalmente, como la similitud del coseno.

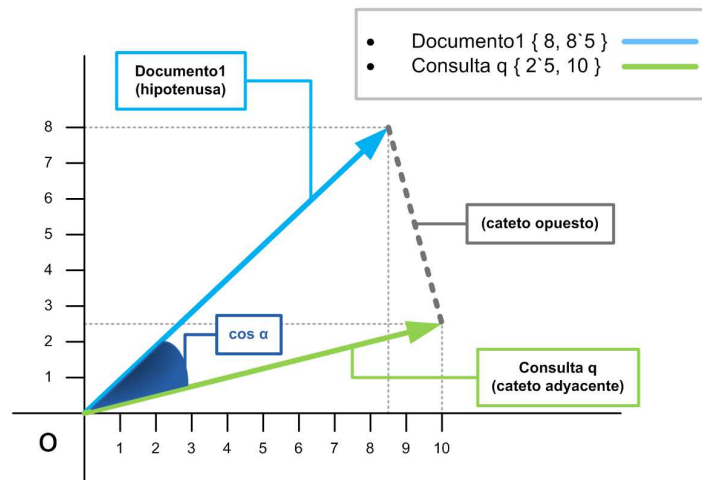


Figura 3.8: Modelo del espacio vectorial

¹La minería de textos es el proceso de analizar colecciones de materiales de texto con el objeto de capturar los temas y conceptos clave y descubrir las relaciones ocultas y las tendencias existentes sin necesidad de conocer las palabras o los términos exactos que los autores han utilizado para expresar dichos conceptos.

²Donde m es la cardinalidad del diccionario (una lista de términos únicos que aparecen en un conjunto de documentos) y $0 \leq w_{ij} \leq 1$ representa la contribución del término t_i para la representación semántica del documento d_j .



Sin embargo, los métodos más complejos son más difíciles de recuperar. Como ejemplo, uno puede considerar la similitud del coseno, que mide el coseno del ángulo entre dos vectores, o equivalentemente el producto de puntos de dos vectores, después de una normalización adecuada.

Por construcción, la similitud de coseno toma en cuenta las palabras comunes contenidas en dos documentos, pero no las relaciones semánticas más complejas, como los sinónimos.

En cambio, una medida de similitud basada en grafos que considera rutas en el documento bipartito *word network* podrá encontrar tales relaciones, siempre y cuando se base en rutas de longitud más de dos entre dos documentos.



Figura 3.9: Red de palabras (*word network*)

Proyección a un grafo unipartito

Otra representación alternativa común de un grafo bipartito es su **proyección a un grafo unipartito**.

En la proyección de un grafo sobre los nodos de la izquierda, solo se mantienen los nodos de la izquierda y los nodos que están conectados cuando tienen un vecino común en el gráfico bipartito original.

La proyección en los nodos de la derecha se define de manera análoga.

Sea $G = (N_1, N_2, A)$ un grafo bipartito, sus proyecciones hacia el lado izquierdo y derecho pueden definirse como los grafos.

Las proyecciones definidas de esta manera se utilizan comúnmente cuando se aplican métodos de análisis de un grafo unipartito a grafos bipartitos.

Entre muchos ejemplos, esta el caso de los tipos de borde que representan una colaboración, cuyo nombre suele comenzar con co-, por ejemplo, la red de co-autoría de científicos o la red de actores co-protagonistas.

Sin embargo, las redes de proyección no rechazan completamente las propiedades de los grafos bipartitos originales.

Por ejemplo, las distribuciones de grados izquierdo y derecho en el gráfico bipartito original se combinarán. Sin embargo, las distribuciones de grado de izquierda y derecha de las redes bipartitas son a menudo muy diferentes, y esto se pierde en la proyección.



Los grafos bipartitos, sin embargo, son mas fáciles de estudiar convirtiéndolos en unipartitos o grafos modo-uno: para ello simplemente se proyectan, eliminando los nodos de uno de los dos tipos, y sustituyéndolos por la relación estar conectados al mismo nodo: dos nodo estarán enlazados si, y solo si, están enlazados al mismo nodo en el grafo bipartito.

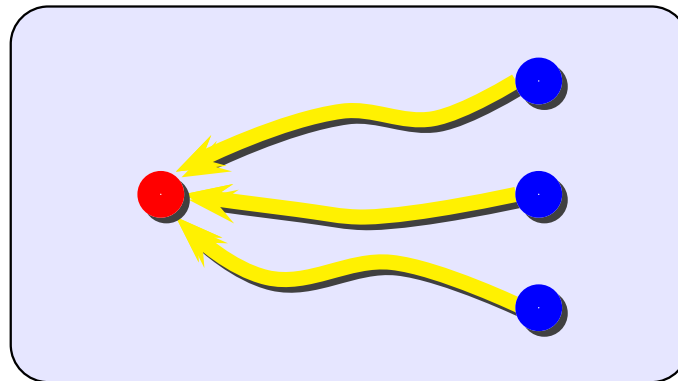


Figura 3.10: Grafo bipartito bicolor

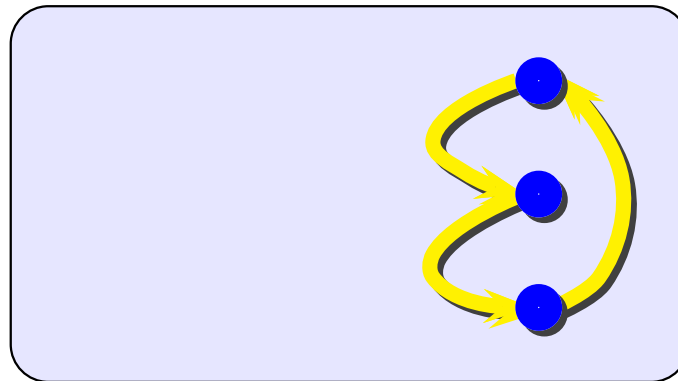


Figura 3.11: Grafo unipartito resultante

El grafo 3.10 se convierte en el grafo que aparece en la figura 3.11 (de modo 1). La distinción entre los grafos de uno u otro tipo es importante, sobre todo, a la hora de calcular un grafo aleatorio que tuviera las mismas propiedades que el grafo estudiado.



3.4. Concepto de pareo en un grafo



Para visualizar en forma clara el concepto de pareo de un grafo bipartito se va a plantear el problema de asignación de tareas para el cual el pareo representa una forma de solución.

Problema de asignación de tareas

Dada una serie de trabajos (tareas) pendientes por realizar y otra de recursos (candidatos) para uno o más de dichos trabajos,

¿Bajo qué condiciones existe una asignación ?

¿Qué asignación de candidatos a tareas permite minimizar los costos totales?

El modelo de asignación es un caso especial del modelo de transporte, en el que los recursos se asignan a las actividades en términos de uno a uno, haciendo notar que la matriz correspondiente debe ser cuadrada. Así entonces cada recurso debe asignarse, de modo único a una actividad particular o asignación.

Se tiene un costo C_{ij} asociado con el recurso i que es asignado a la tarea j , de modo que el objetivo es determinar en que forma deben realizarse todas las asignaciones para minimizar el costo total.

El problema de asignación puede transformarse en un problema de transporte al considerar a los trabajadores como orígenes y a las tareas como destinos con todos los suministros y demandas igual a uno.

Representación mediante grafos

El problema de la asignación de tareas también puede expresarse en términos de Teoría de Grafos como: ¿Bajo qué condiciones un grafo bipartito G dado posee un subgrafo 1-regular que incluya a todos los nodos de uno de los subconjuntos partitos, el que representa a los candidatos (o bien a los trabajos)? ¿Cuál es el tamaño máximo de un subgrafo 1-regular de un grafo bipartito?

Modelado del problema

En un grafo G bipartito ponderado, si el candidato (R) se le asigna al recurso tarea (T), entonces el peso de la arco RT , $w_{(RT)}$, en G es una medida del beneficio que se obtiene asignando al candidato R para emplear el recurso T .

El problema de encontrar una asignación de candidatos a trabajos vacantes que produzca un beneficio máximo es equivalente a encontrar un subgrafo 1-regular H de G tal que la suma de los pesos de los arcos de H sea máximo.



Ejemplos

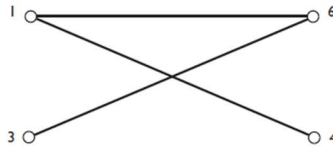


Figura 3.12: En este caso: NO es emparejamiento, ya que salen dos arcos de 1

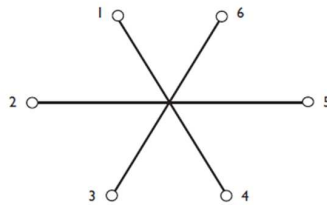


Figura 3.13: SÍ es un emparejamiento, ya que no sale más de un arco de un mismo nodo

Un emparejamiento en $G = (S \cup T, A)$ es un conjunto E de arcos tales que ningún par de ellas tiene un nodo en común.

Si el emparejamiento es del mayor número posible de arcos se denomina emparejamiento máximo o de cardinal máximo.

Un nodo se dice **expuesto** (o no cubierto) por un emparejamiento E si no es punto extremo de ninguna arco de E .

Ser emparejamiento **máximo** no significa que todos los nodos estén emparejados con otro, sino que están emparejados los máximos posibles.

Por ejemplo, si S tiene 5 nodos y T tiene 7, a lo más puede obtenerse un emparejamiento de tamaño 5, pero también es posible que no puedan emparejarse los 5 nodos:

En el grafo G_1 , en la fig.3.14, el emparejamiento máximo es de tamaño 3, mientras que en G_2 , en la fig.3.15, es de tamaño 4.

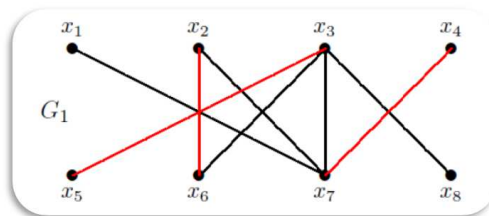
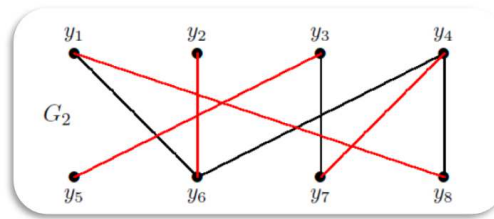


Figura 3.14: Grafo bipartito G_1

Figura 3.15: Grafo bipartito G_2

Un emparejamiento como el de G_2 (fig.3.15), que no deja ningún borde expuesto, se dice que es un **emparejamiento completo** (evidentemente, un emparejamiento completo es máximo y sólo puede darse si los dos conjuntos tienen el mismo número de nodos).

Los nodos x_1 y x_8 no están cubiertos por el emparejamiento dado en G_1 (fig.3.14), es decir, son expuestos.

La manera de proceder para obtener un emparejamiento máximo, es ir incrementando paso a paso el cardinal del emparejamiento hasta hacerlo máximo. Para esta idea es fundamental el concepto de **camino aumentador**.

Algoritmo de emparejamiento

Sea $G = (S \cup T, A)$ un grafo bipartito de n nodos y E un emparejamiento en G (puede ser inicialmente $E = \emptyset$).

Denotemos los nodos por $1, 2, \dots, n$.

1. Si en S no hay ningún borde expuesto, E es máximo.
 - En caso contrario, etiquetar cada nodo expuesto de S por 0.
2. Para cada nodo $j \in S$ y arco $j, k \notin E$, etiquetar el nodo k con j , a menos que ya esté etiquetado.
3. Para cada nodo $k \in T$ cubierto por E , etiquetar al nodo j por k , donde j es el nodo de S extremo de la única arco $\{j, k\} \in E$.
4. Recorrer hacia atrás los caminos alternantes que acaban en un nodo expuesto en T , usando las etiquetas sobre los nodos.
5. Si en el paso anterior, ninguno de los caminos alternantes es de aumento, entonces E es máximo.
 - En caso contrario, aumentar E usando un camino aumentador y eliminar todas las etiquetas y volver al Paso 1.



3.5. Modelando problemas reales con grafos bipartitos

Arribos de aviones

Por razones obvias de seguridad, las autoridades aeroportuarias deben garantizar que el número de aviones de aterrizaje en un intervalo de tiempo determinado sea inferior a la capacidad del aeropuerto.

Por lo tanto, los aeropuertos tienen intervalos de tiempo (frangas horarias) que se asignan inicialmente a las aeronaves según su programa de llegadas



Figura 3.16: Arribos de aviones

Una aeronave solo puede asignarse a una pista que sea compatible con su hora de llegada.

Sin embargo, si se retrasa un vuelo, los controladores de operación de la línea aérea deben asignar una nueva pista a través del sistema de información que administra estos intercambios.

Las estrictas regulaciones que deben respetar estos sistema hacen que sólo sean posibles dos operaciones.

Se asigna una pista disponible a la aeronave A (**Regla 1**), o una pista S que ya está asignada a otra aeronave B se reasigna a A mientras que a B se le asigna una pista disponible S_0 (**Regla 2**).

En ambos casos, la pista S debe ser compatible con el programa de A y en el segundo caso, S y S_0 deben ser compatibles con la de B .

Si varios aeroplanos se atrasan y pierden sus pistas, la resolución de estos problemas es difícil.

Ya que los controladores aéreos no tienen las herramientas de sistema para realizar estos cambios y deben garantizar *a mano* que todas las aeronaves obtendrán un espacio.

Cada avión está vinculado a las pistas que son compatibles con él. Inicialmente, los Slots 1,2,4 y 6 se asignan a los aviones A , C , B y D , respectivamente.



Los bordes de coincidencia se representan en negritas. Digamos que, después de que A y D se hayan retrasado, ya no son compatibles con las pistas 1 y 6.

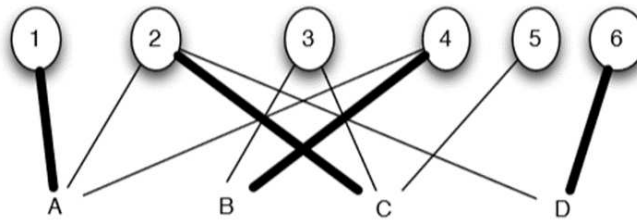


Figura 3.17: Asignación actual de slots

Por lo tanto, la configuración se convierte en la que se muestra en la Fig.3.18(a). Desde la Configuración (a), si la pista 2 se reasigna a A y la pista 3 se asigna a C (es decir, la Regla 2 se aplica a A y C), entonces llegamos a Conf.(b) donde no es posible ninguna modificación permitida (no hay Regla puede ser aplicado).

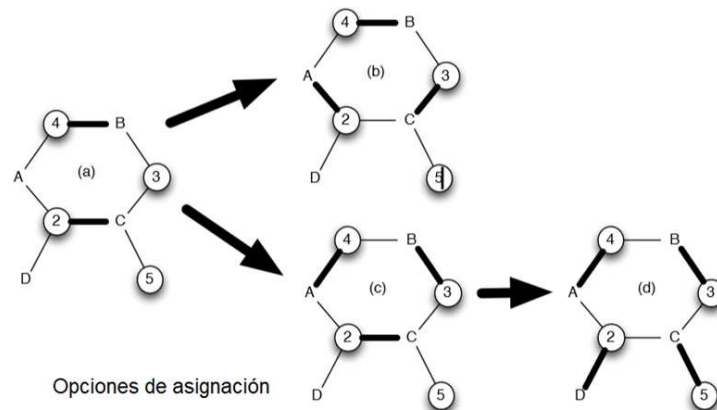


Figura 3.18: Opciones de asignación

Otra solución sería aplicar la Regla 2 a A y B (A obtiene la pista 4 y B obtiene la pista 3), alcanzando Conf. (c), y luego aplicar la Regla 2 a C y D (C obtiene la pista 5 y D obtiene la pista 2), llegando a la Conf. (d) donde a todas las aeronaves se les asigna un espacio.

Pareo de grafos

El problema de la reasignación de pistas puede, por supuesto, ser modelado como un problema de emparejamiento de grafos.

Un pareo M que coincide con E de un gráfico $G = (N, E)$ es un conjunto de bordes separados por pares.



El tamaño máximo de una coincidencia en G se denota por $\mu(G)$. El problema de calcular una coincidencia máxima ha sido ampliamente estudiado y es bien sabido que puede resolverse en tiempo polinomial.

Un ingrediente clave en la mayoría de los trabajos sobre emparejamientos es la noción de **camino aumentador**.

Una ruta $P = (v_0, \dots, v_k)$ de G es una secuencia de nodos distintos de pares de manera que $e_i = \{v_i, v_i + 1\} \in E$ para cada $0 = i < k$.

La ruta P se dice aumento de M si v_0 y v_k están expuestos (no inciden en un borde de M) y, para cualquier $0 = i < k$, $e_i \in M$ si y solo si i es impar.

Cuando se pasa de una M coincidente a la $M \Delta E(P)$ coincidente, se dice que la ruta P de aumento de M está aumentada.

Algunos algoritmos que calculan las coincidencias máximas en gráficos se basan en aumentar dichas rutas en el orden no decreciente de sus longitudes.

El problema de la reasignación de pistas descrito anteriormente se puede modelar de la siguiente manera:

Sea $G = (X \cup Y, E)$ un grafo bipartito.

X representa el conjunto de aviones y Y representa el conjunto de pistas.

Hay un arco entre $a \in X$ y $s \in Y$ si la pista s es compatible con el horario de la aeronave A .

Sea M una coincidencia de G que corresponda a una asignación válida preestablecida de algunas pistas a algunas aeronaves.

El problema de la reasignación de pistas es equivalente a calcular una coincidencia máxima que se puede obtener de M aumentando sólo las rutas de longitud a lo sumo 3.

Por ejemplo, el primer escenario en el ejemplo anterior consiste en aumentar la ruta $(A2C3)$. Al llegar a Conf. (b), no hay rutas de aumento de longitud a lo sumo 3. El segundo escenario consiste en aumentar primero la ruta $(A4B3)$ (llegando a Conf. (c)) y luego a la ruta $(D2C5)$ para llegar a Conf. (d).

Este ejemplo ya sugiere que delimitar la longitud de las rutas aumentadas puede aumentar la dificultad del Problema de coincidencia máxima, ya que el orden en que se aumentan las rutas se vuelve importante.

Cuando $k \geq 3$, la dificultad surge del hecho de que el orden en que se aumentan los caminos es importante.

Este hecho se ilustra en la Fig.3.18, donde aumentar primero la trayectoria $(A2C3)$ conduce a una configuración de punto muerto no óptima.

Además, el orden en que se aumentan las rutas tiene un impacto en la creación o no creación de nuevas rutas de aumento de longitud como máximo k .

Por ejemplo, para $k = 5$, consideremos el grafo de la Fig.3.19 que consta de una ruta (v_1, \dots, v_7) más tres bordes $\{v_5, v_8\}$, $\{v_8, v_9\}$ y $\{v_9, v_{10}\}$.

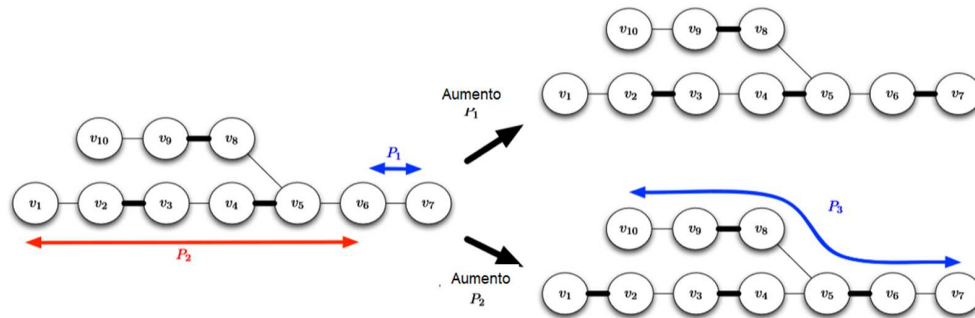


Figura 3.19: Grafo de reasignación

La coincidencia inicial es $\{\{v_2, v_3\}, \{v_4, v_5\}, \{v_8, v_9\}\}$. Inicialmente, hay dos rutas de aumento de longitud a lo sumo 5: $P_1 = (v_6, v_7)$ y $P_2 = (v_1, \dots, v_6)$.

Si P_1 se aumenta primero, entonces no quedan más rutas de aumento de longitud a lo sumo 5.

Sin embargo, aunque aumentar P_2 destruye la ruta P_1 , también crea una nueva ruta de aumento $P_3 = (v_{10}, v_9, v_8, v_5, v_6, v_7)$ que se puede aumentar.

Problema de ruteo de vehículos

Un problema de rutas de vehículos consiste en determinar las rutas de un conjunto (o flotilla) de vehículos que deben iniciar un recorrido (y finalizarlo) en los almacenes (o depósitos) para atender la demanda de servicio de un conjunto disperso de clientes sobre una red.



Figura 3.20: Problema de ruteo de vehículos

El transporte es uno de los sectores que más aporta a la generación de riqueza en el mundo.



El transporte por carretera muestra un comportamiento netamente superior al resto de los modos y actividades de transporte (ferroviario, marítimo y aéreo) en cuanto a generación de valor.

En este trabajo se analizan todos los factores que se tienen en cuenta a la hora de elaborar las rutas diarias de cada operario de un vehículo: la planificación inicial de las rutas llevada a cabo por el jefe de taller y la encargada de control y gestión de los equipos, así como los cambios llevados a cabo por los operarios en una ruta decidiendo el orden de las visitas en base a diferentes factores no considerados en la determinación de la ruta inicial, tales como: el consumo diario actual del cliente, la zona geográfica, las necesidades específicas del cliente, la comodidad, el tráfico, etc..

Trata una situación en la cual se envía un bien desde uno o varios puntos de origen hasta uno o varios puntos de destino con el objetivo de determinar la cantidad enviada, satisfaciendo al mismo tiempo las restricciones de la oferta y la demanda, y minimizando el costo total del envío.

Este problema también se caracteriza por suponer que el costo de envío en una ruta determinada es directamente proporcional al número de unidades enviadas en esa ruta.

Las características diferentes de los clientes, la demanda, los almacenes y los vehículos, así como de las restricciones operativas sobre las rutas, horarios, etc. dan lugar a gran número de variantes del problema.

El problema específico surge de una empresa perteneciente al sector de la distribución automática, cuya actividad empresarial es distribución y abastecimiento de máquinas vending y de agua refrigerada.

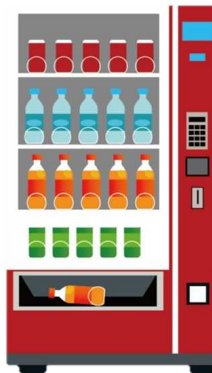


Figura 3.21: Distribución y abastecimiento de máquinas vending

Esta actividad pone a disposición del consumidor una amplia gama de productos a través de un nuevo modelo de distribución, constituyendo al tiempo, un punto de vista diferenciado de lo tradicional, puesto que no requiere la presencia humana de un vendedor.



La empresa objeto de estudio distribuye sus máquinas principalmente en entornos laborales, centros públicos y privados y en empresas dedicadas al sector servicios. Cualquier empresa del mercado se enfrenta a variables externas que afectan su gestión.

En el caso de estudio, destaca la tendencia en los precios de los recursos energéticos, ya que la actividad de la empresa supone un desplazamiento continuo de todos los operarios a todos los clientes.

La planificación actual de las rutas de trabajo depende de los siguientes factores: localización del cliente, consumo mensual y tipo de máquina. Posteriormente, el operario realiza cambios en esta ruta basándose en unos criterios previstos y que dan lugar a la ruta replanificada: jornada laboral de los clientes, media mensual de consumo, ubicación de las máquinas y dimensión de la posición, ampliación del número de máquinas instaladas en un mismo cliente, modificación de la capacidad de la máquina, nueva línea de negocio de la empresa, periodo de caducidad de los productos, etc..

También se consideran una serie de factores imprevistos como:

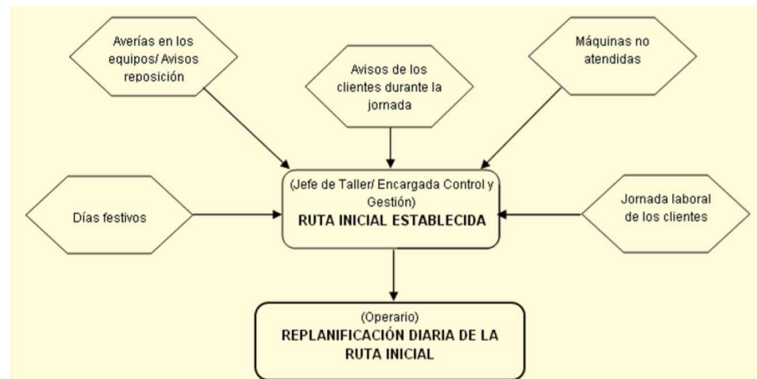


Figura 3.22: Replanificación de las rutas iniciales

El grafo representativo de este problema se muestra a continuación:

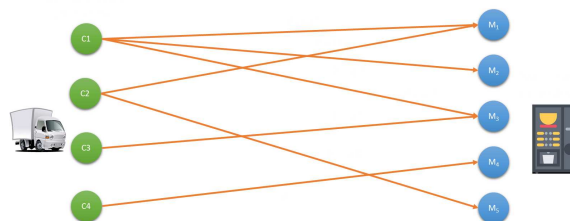


Figura 3.23: Grafo bipartito



Para determinar el costo asociado a cada arco se define la siguiente función:

$$f_{ij} = Km_{ij} \times Cu + t_{ij} \times Cm + CTributos$$

Donde:

- Km representa una matriz de distancia (en kms)
- Cu matriz de costo del recurso energético
- t matriz de tiempos de recorrido (en minutos)
- Cm matriz de costo de las horas trabajadas por los operarios (en moneda)
- CT matriz de costo de la circulación de los camiones (costas e impuestos al transporte)

Aunque estos son los componentes fundamentales de los costos directos de las empresas de transporte, el modelo se podría extender fácilmente introduciendo otras variables como son los gastos en neumáticos, mantenimiento, reparaciones y amortización y financiación de los vehículos.

Desde el punto de vista de los costos, este estudio proporciona un total de 16 valores mínimos.

Esto significa que de las veinte jornadas diarias analizadas que componen cada mes estudiado (Semana 1-Lunes ó $S1 - L$,...Semana 4-Viernes ó $S4 - N$) la aplicación ha obtenido en dieciséis de éstas costos mínimos óptimos. Por tanto, el costo global de una ruta disminuye dado que se repite la misma secuencia de visitas durante la primera y tercera semana, y durante la segunda y la cuarta semana para todos los meses analizados.

Estos resultados comparados con el único valor mínimo que ha generado la ruta inicial confirman que las secuencias provistas por dicha aplicación ayudan a reducir el costo a la empresa.

La diferencia entre los valores mínimos obtenidos frente a los máximos, suponen un ahorro anual total de 2.610 unidades para esta ruta.

Al comparar la ruta inicial con la ruta óptima obtenida en este estudio, anualmente la empresa ahorraría la cantidad de 984,82 unidades para la ruta estudiada siempre y cuando se modificara la ruta inicial y la adaptara a los resultados obtenidos.



3.6. Problemas de Pareo en grafos. Pareos perfectos y maximales

Pareos perfectos

Un **emparejamiento perfecto** de un grafo es una colección de arcos separados que cubren todos los nodos. Por ejemplo, el gráfico que se muestra en la Fig.3.24(a) tiene un total de tres coincidencias perfectas, como se indica en la config.(b).

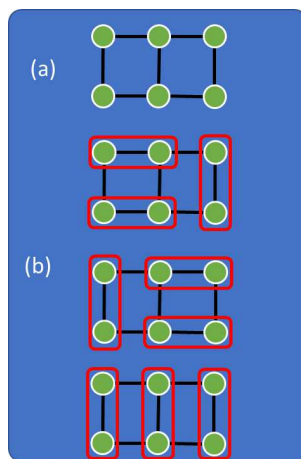


Figura 3.24: Grafo de 3 emparejamientos perfectos

Se va a denotar el número de coincidencias perfectas de un gráfico G por $M_p(G)$. Veamos ahora un ejemplo más grande. Supongamos que queremos encontrar todos los emparejamientos perfectos del gráfico de 12 nodos que se muestra a la izquierda de la Fig.3.25.

Se dividen en dos posibilidades según si el nodo superior izquierdo se empareja horizontal o verticalmente.

En el primer caso, otros tres bordes (rodeados por líneas de puntos en la Fig.3.25) son obligados a formar parte de la coincidencia perfecta, y lo que queda sin cubrir es solo un ciclo de 4 ciclos, que claramente tiene dos coincidencias perfectas.

El último caso se ramifica en dos, según que el nodo superior izquierdo coincida horizontal o verticalmente.

Ahora, la instancia con emparejamiento vertical obliga a 2 bordes, dejando dos formas de terminar este subcaso.

Las ramas alternativas coincidentes horizontalmente en dos subramas más; Se ve fácilmente que cada uno de ellos tiene exactamente dos formas de completarlos para obtener una coincidencia perfecta (ver la figura).

Por lo tanto, el número total de coincidencias perfectas de este gráfico es 8.

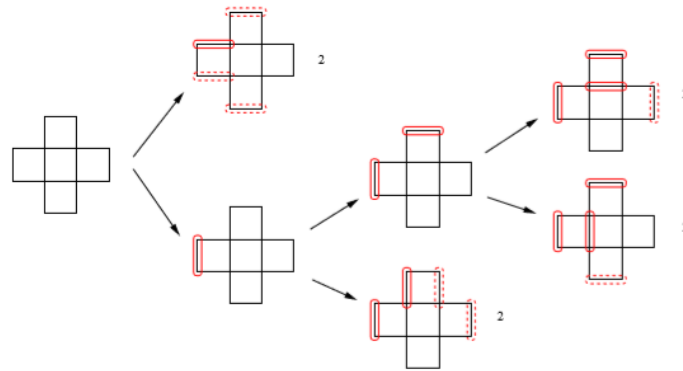


Figura 3.25: Grafo con 8 emparejamientos perfectos

Un último ejemplo hecho *a mano* es el gráfico de cuadrícula 4×4 , ilustrado en la Fig.3.26. Nos dividimos en dos de acuerdo a que el nodo superior izquierdo coincide horizontal o verticalmente; Por simetría, los dos casos se pueden extender a un emparejamiento perfecto en la misma cantidad de formas.

La Fig.3.26 describe la rama superior a medida que se divide en subdivisiones, dependiendo de cómo coincidan ciertos nodos. El número de formas de terminar los emparejamientos parciales en las puntas del árbol se encuentra fácilmente como se indica en la imagen. Obtenemos $M(G_4) = 36$.

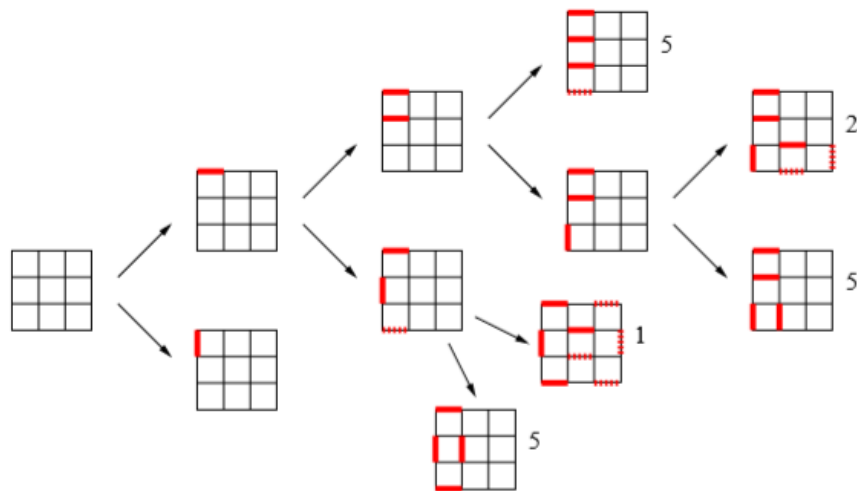


Figura 3.26: Grafo con 36 emparejamientos perfectos



Pareos maximales

Un **emparejamiento maximal** es un emparejamiento que no puede crecer por agregado de un arco. Cumple la propiedad de que al añadir algún arco que no pertenece al emparejamiento, se deshace el emparejamiento.

Definición (Emparejamiento maximal)

Si $G = (N, E)$ es un grafo bipartito con N dividido como $X \cup Y$, un emparejamiento maximal en G es aquel que relaciona el mayor número posible de nodos en X con los nodos en Y .

Un *emparejamiento máximo* es un emparejamiento que contiene el número máximo posible de arcos. Por lo tanto, no puede estar contenido en otro de cardinal mayor.

Atendiendo a las definiciones anteriores se puede formular los siguientes teoremas:

- Puede haber muchos emparejamientos máximos.
- Los emparejamientos máximos son maximales, pero no todos los emparejamientos maximales son máximos.
- Cada emparejamiento perfecto es máximo y maximal.

Los problemas de emparejamiento guardan una importante relación con los grafos bipartitos. Como ya se estableció un grafo $G = (N, A)$ se dice *bipartito* si el conjunto de nodos N puede separarse en dos conjuntos disjuntos $N = X \cup Y$ de tal manera que los arcos del grafo unen siempre un nodo de X con uno de Y , es decir, no hay arcos entre los nodos de X ni entre los nodos de Y .

Para obtener un emparejamiento máximo bipartito, es necesario definir previamente un par de conceptos:

- Un **camino alterno** es un camino en el cual sus arcos alternativamente pertenecen y no pertenecen al emparejamiento.
- Un **camino incremento** es un camino alterno que comienza y termina en un nodo libre.

La manera de proceder para obtener un emparejamiento máximo bipartito consiste en ir incrementando paso a paso el cardinal del emparejamiento mediante caminos incremento hasta obtener un emparejamiento máximo.

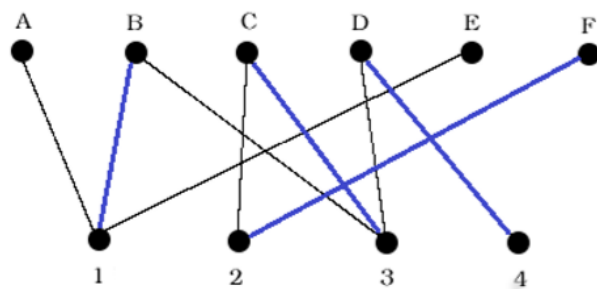


Figura 3.27: Emparejamiento máximo bipartito



3.7. Aplicaciones

El problema de los Matrimonios Estables

Este es un problema de asignación bipartito que parte de dos conjuntos de personas que tienen el mismo cardinal, hombres y mujeres, donde cada persona de cada grupo tiene una lista de preferencias en la que ordena a los miembros del sexo opuesto.

El problema consiste por tanto en encontrar un emparejamiento estable entre ambos grupos de personas.

La estabilidad de este problema de asignación es muy importante, ya que no se conforma con que un hombre y una mujer estén emparejadas con otra persona si ambos preferirían estar juntos antes que con sus parejas actuales.

A pesar de que este problema ha sido muy estudiado a lo largo de las últimas décadas, los primeros en encontrar una solución para el mismo fueron D.Gale y G.S.Sapley, quienes en 1962 plantearon el llamado **Algoritmo de Gale-Shapley**. Con este algoritmo se demostró que, si el número de hombres y mujeres es el mismo, siempre existe un emparejamiento estable entre ellos.

Este problema se puede definir en términos de grafos, puesto que se parte de dos conjuntos disjuntos del mismo cardinal, $H = \{h_1, h_2, \dots, h_n\}$ y $M = m_1, m_2, \dots, m_n$, que conformarán el conjunto de nodos del grafo bipartito ($V = H \cup M$, con $|H| = |M|$).

En términos de matrimonios, el concepto de **pareja bloqueante** se puede formular de la siguiente manera.

Si hay dos matrimonios (h_i, m_i) y (h_j, m_j) tales que h_i prefiere estar antes con m_j que con m_i , y m_j prefiere estar antes con h_i que con h_j , entonces (h_i, m_j) es una pareja bloqueante, ya que tanto h_i como m_j se divorciarán de sus parejas para estar juntos.

En esta definición se basa el concepto clave de la estabilidad del emparejamiento.

Ejemplo

Sean los conjuntos $H = \{h_1, h_2, h_3\}$ y $M = \{m_1, m_2, m_3\}$ de hombres y mujeres respectivamente. Se cuenta también con la lista de preferencias de los hombres sobre las mujeres y de las mujeres sobre los hombres, mostradas a continuación:

h_1	m_1	m_2	m_3
h_2	m_1	m_3	m_2
h_3	m_2	m_1	m_3

Tabla 3.1: Preferencias hombre



m_1	h_1	h_3	h_2
m_2	h_3	h_1	h_2
m_3	h_3	h_2	h_1

Tabla 3.2: Preferencias mujer

En esta situación, el emparejamiento $(h_1, m_1), (h_2, m_3), (h_3, m_2)$ sería estable, ya que no hay parejas bloqueantes que prefieran estar juntas antes que con sus respectivas parejas.

Así, los pares $(h_1, m_1), (h_2, m_3), (h_3, m_2)$ son todos ellos parejas válidas porque existe un emparejamiento estable, en este caso el formado por ellas tres juntas, para el cual son parejas.

Sin embargo, el emparejamiento $(h_1, m_2), (h_2, m_1), (h_3, m_3)$ no sería estable, ya que el par (h_1, m_1) sería una pareja bloqueante, puesto que h_1 prefiere estar con m_1 antes que con m_2 , y m_1 prefiere estar con h_1 antes que con h_2 .

Al principio, todos los hombres y las mujeres están desemparejados. Cada h_1 no emparejado le propone ser su pareja al nodo m_j que más prefiere según su ordenación (siempre que ese nodo no le haya rechazado ya).

Cada nodo m_j que haya recibido alguna propuesta en el paso anterior:

- Si no estaba emparejado, se empareja con el nodo h_i de los que tiene propuesta que más prefiera según su ordenación, y deja a los demás como nodos libres del emparejamiento.
- Si ya estaba emparejado, se empareja con el nodo h_i que más prefiere de entre todas sus propuestas (incluido su pareja actual), y deja a los demás como nodos libres del emparejamiento.
- Se repiten los pasos 1 y 2 hasta que todos los nodos h_i estén emparejados con un nodo m_j .

Algoritmo (Gale-Shapley)

1. Cada mujer propone su primera elección.
2. Los hombres con dos o más propuestas responden rechazando a todas menos a la más favorable.
3. Las mujeres rechazadas proponen su segunda elección. Las que no fueron rechazadas continúan con su propuesta.
4. Se repiten los pasos 2 y 3 hasta que ninguna propuesta sea rechazada



```
algorithm stable_matching is
  Initialize  $m \in M$  and  $w \in W$  to free
  while  $\exists$  free man  $m$  who has a woman  $w$  to propose to do
     $w :=$  first woman on  $m$ 's list to whom  $m$  has not yet proposed
    if  $\exists$  some pair  $(m', w)$  then
      if  $w$  prefers  $m$  to  $m'$  then
         $m'$  becomes free
         $(m, w)$  become engaged
      end if
    else
       $(m, w)$  become engaged
    end if
  repeat
```

Figura 3.28: Pseudocódigo Algoritmo de Gale-Shapley

Bibliografía

- [Diccionario Larousse, 2003] “*Diccionario Larousse, Edición Premium*”, EDICIONES LAROUSSE MÉXICO y SPS EDITORIAL BARCELONA, (2003).
- [DRALE] “*Diccionario de la Real Academia de la Lengua Española*”, versión on-line (2017).
- [Dijkstra, 1959] Dijkstra, E. W. “*A note on two problems in connexion with graphs*. Numerische Mathematik. 1: 269-271, (1959).
- [Johnson 1963] Johnson, Selmer M., “*Generation of permutations by adjacent transposition*”, Mathematics of Computation, 17: 282-285, (1963).
- [Heap, 1963] Heap, B R., “*Permutations by Interchanges*”, Computer Journal, 6, 293-4, (1963).
- [Kruskal, 1956] Kruskal, J. B., “*On the shortest spanning subtree of a graph and the traveling salesman problem*. Proceedings of the American Mathematical Society, (1956).
- [Prim, 1957] Prim, R. C., “*Shortest connection networks And some generalizations*, Bell System Technical Journal, 36 (6), November (1957).
- [Sedgewick, 1977] Sedgewick, R., “*Permutations Generation Methods*”, ACM Computing Surveys, Vol. 9, No. 2, Junio (1977).
- [Trotter, 1959] Trotter, H. F., “*On the product of semi-groups of operators*”, Proceedings of the American Mathematical Society, 1:4, 545-551, (1959).
- [Warshall, 1962] Warshall, F. R., “*Algorithm 97: Shortest Path*. Communications of the ACM. 5 (6): 345., June (1962).
- [Wilhelm Leibnitz, 1666] Leibniz, Gottfried W., “*Dissertatio de arte combinatoria*”, (versión extendida de su 1a Tesis Doctoral), Leipzig, (1666).