TEORÍA DE ALGORITMOS 1

Algoritmos naive y por fuerza bruta

Por: ING. VÍCTOR DANIEL PODBEREZSKI
vpodberezski@fi.uba.ar

1. Algoritmos Ingenuos

Se considera como un algoritmo de **tipo ingenuo** (naive) a aquel que resuelve un problema utilizando alguna propiedad básica del mismo y sin valerse de estructuras ni estrategias complejas para hacerlo. En ocasiones se los suele ubicar dentro de la familia de algoritmos por fuerza bruta. Sin embargo, no es necesariamente el caso. Muchos algoritmos de otras metodologías se pueden implementar de forma naive.

Un típico ejemplo de algoritmo naive corresponde al tradicional algoritmo de multiplicación de matrices. En el problema de **multiplicación de dos matrices** partimos de 2 matrices cuadradas (aunque se puede generalizar para matrices rectangulares) y debemos calcular su producto. Formalizando el problema, anunciaremos:

Multiplicación de dos matrices cuadradas de nxn
Dado A y B matrices de nxn. Deseamos calcular la matriz resultante de multiplicar A por B.

Partimos de dos matrices A y B ambas de nxn. Realizaremos AxB y llamaremos C a su resultado. Este será también una matriz cuadrada de nxn. Utilizando la misma definición de la multiplicación cada celda c de fila i y columna j se calculará como $C_{ij} = \sum_{k=1}^{n} A_{ik} x\, B_{kj}$.

Podemos resumir el pseudocódigo de la operación como:

Multiplicación de matrices: Algoritmo naive
Sea X y Y dos matrices cuadradas de nxn Sea Z la matriz resultante de nxn

```
Desde i=1 a n
        Desde j=1 a n
                Z[i,j]=0
                Desde k=1 a n
                        Z[i,j] = Z[i,j]  + X[i,k] * Y[k,j]
```

En este algoritmo naive podemos ver n^3 multiplicaciones y una cantidad similar de adiciones. Es decir que se puede expresar su complejidad temporal como $O(n^3)$. Corresponde a un algoritmo bueno y nuestra experiencia con esta operatoria nos dice que corresponde a un problema tratable. Por mucho tiempo este fue el mejor procedimiento conocido en relación a la complejidad temporal. Sin embargo a fines de la década del 60 se presentó un algoritmo superador utilizando la metodología de división y conquista. Lo analizaremos en el módulo dedicado a ese tema.

En el mismo sentido la **multiplicación de números enteros grandes** (de muchos dígitos) que no pueden realizarse como una operación unitaria por una computadora tiene una manera de resolverse "naive". El algoritmo corresponde al aprendido en el nivel inicial del sistema educativo. Formalizamos el problema como:

Multiplicación de dos números de n bits cada uno
Dado A y B números de n bits. Deseamos hallar el número resultante de multiplicar A por B.

Utilizamos números de igual longitud, sin embargo el mismo proceso se puede realizar con números de longitud diferente, modificando levemente el algoritmo. El resultado será un número de como mucho 2n longitud.

Multiplicación de números enteros: Algoritmo naive

```
Sea A y B dos números de n dígitos
Sea C el número resultante de 2n dígitos
Sea base la base en la que están representados los números

Desde i=1 a n
        Sea resto = 0
        Desde j=1 a n
```

```
            C[i + j - 1] += resto + a[j] * b[i]
            resto = C[j + i - 1] / base
            C[j + i - 1] = C[j + i - 1] mod base
    C[i + n] = resto
```

Consideramos cada una de las operaciones aritméticas O(1). Vemos que la existencia de dos anidados acotados por la cantidad de dígito nos da una complejidad global de $O(n^2)$. Es un algoritmo bueno. Requiere un O(n) de espacio adicional para almacenar el resultado de la operación. Este algoritmo no es el de menor complejidad disponible, veremos posteriormente el algoritmo de Karatsuba que resulta superador utilizando división y conquista.

Otro problema que podemos utilizar de ejemplo es el **problema de la subcadena común más larga** (Longest Common Substring). En este partimos de dos secuencias de elementos de longitud n y queremos obtener cual corresponde a la sub secuencia contigua de mayor longitud iguales que contienen. Lo enunciamos como:

Problema de la subcadena común más larga
Dadas las secuencias A y B que contienen n elementos cada una. Queremos conocer cual corresponde a la subsecuencia contigua de mayor longitud incluida en ambas.

Vemos un simple ejemplo para que sea más claro el problema. Usaremos que los elementos son caracteres y que cada secuencia tiene una longitud de 25. La secuencias que utilizaremos serán A='dabalearrozalazorraelabad' y B='elrozabalaparrayelaladaba'. Buscando podemos encontrar que "el" corresponde a una secuencia de longitud 2 que se encuentra en ambas secuencias. También podemos encontrar la secuencia de longitud 3 en ambos "ela". Sin embargo la de mayor longitud corresponde a 'daba' con longitud 4.

Un algoritmo naive para resolver el problema tomaría una de las secuencias y por cada posición de la misma verifica si alguna de las posiciones de la otra secuencia contiene el mismo elemento. De coincidir comienza a verificar por los siguientes elementos hasta que finalice alguna de las secuencias o la coincidencia finalice. El siguiente es un pseudocódigo que utiliza la estrategia descripta:

Problema de la subcadena común más larga: Algoritmo naive

```
Sea A y B dos secuencias de n elementos

Sea posA,posB las posiciones de las máximas longitudes
Sea longitud=0 la longitud máxima encontrada
Desde i=1 a n
        Desde j=1 a n
                l=1
                Mientras A[i..(i+l-1)]==B[j..(j+l-1)] y l<n-min(i,j)
                        Si longitud<l
                                longitud=l
                                posA=i
                                posB=j
                        l++
```

La solución naive corresponde a una propuesta polinomial $O(n^3)$. Utilizando programación dinámica se puede lograr una complejidad menor.

Ciertos autores consideran algunos métodos de ordenamiento como pertenecientes a la categoría naive: ordenamiento por selección, ordenamiento de burbuja y ordenamiento por inserción. Todos estos coinciden en basarse en una propiedad de ordenamiento de elementos simples, no utilizar estructuras ni estrategias complejas y tener una complejidad temporal de $O(n^2)$. No ahondaremos en ellos dado que consideramos que son conocidos por el lector. Pero realizaremos un breve repaso. Supongamos que queremos ordenar los elementos de menor a mayor.

En primer lugar, el ordenamiento **por selección** (Selection Sort) considera que en un listado ordenado de “n” elementos en la primera posición debe estar el más pequeño. Por lo tanto recorre todo los elementos obteniendo a este y ubicándolo en este primer lugar. Luego repite el proceso para n-1 elementos buscando el menor valor entre los que quedaron y repite hasta finalizar. No se utiliza ninguna estructura adicional en el ordenamiento excepto un registro de donde está el menor elemento encontrado y una función de intercambio. Algunos autores incluyen a este algoritmo también en la metodología codiciosa o greedy.

El **ordenamiento de burbuja** (Bubble Sort) considera que en un listado ordenado si comparamos dos elementos contiguos el primero de ellos debe ser menor al segundo. Si no lo es entonces procede a intercambiarlos de lugar. El algoritmo procede a realizar la

comparación para cada par contiguo de elementos iniciando desde el primero hasta el último. Mediante este proceso un elemento puede adelantarse como mucho una posición. En el peor de los casos el menor de los elementos se encuentra en la última posición al iniciar el ordenamiento. En ese caso para que arribe al primer lugar se debe realizar n veces el proceso descripto. El resto de los elementos requiere igual o menos movimientos para ubicarse en el lugar que le corresponde. Este algoritmo tampoco requiere estructuras adicionales.

El **ordenamiento por inserción** (insertion sort) considera que agregar un elemento a una lista previamente ordenada simplemente implica buscar en qué posición agregarla. Este algoritmo comienza con una lista vacía inicial (y considerada ordenada) y comienza a tomar de a uno los elementos e insertarlos de forma ordenada. Finaliza luego de ubicar el último de los elementos. Por cada inserción se debe buscar la posición del elemento y mover todos los elementos posteriores en un lugar para intercalar. Como los anteriores, no requiere estructuras adicionales ni estrategias complejas. Este método tiene como particularidad que se puede utilizar de forma adecuada si los elementos a ordenar llegan de forma asincrónica. Se puede considerar al mismo como dentro de la metodología online y de mejora incremental.

Existe una gran variedad adicional de problemas naive, veremos algunos más adelante junto a problemas de otras metodologías. Nos sirven como puntapié de otras soluciones o como material de comparación

2. Algoritmos por fuerza bruta

Cuando se habla de resolver un problema por "fuerza bruta" nos hacemos la idea de un proceso arduo "manual" o "físico" dictado por una estrategia de resolución construida mediante un pobre o mínimo trabajo "mental".

Dentro de los problemas computacionales existen una gran cantidad para los que se han propuesto algoritmos de resolución mediante "fuerza bruta". Existen dentro de esta denominación una variedad de metodologías que terminan siendo catalogadas bajo esta denominación. Algunas de estas son similares entre sí o engloban a un subconjunto de problemas con ciertas características comunes. Entre estas denominaciones podemos

encontrar: búsqueda exhaustiva, Generar y probar, vuelta atrás (backtracking), ramificar y probar (branch and bound), entre otros.

Si hay algo que unifica a estos algoritmos es que, por regla general, corresponden a tipos de solución fáciles de entender y fáciles de implementar. La característica que le da nombre es que, en su peor caso, en base a la definición del problema terminan probando todas las posibles soluciones posibles hasta encontrar la o las óptimas.

En este tipo de problemas es fundamental definir el espacio de búsqueda y una estrategia para recorrerlo. Un mismo problema muchas veces puede ser resuelto utilizando diferentes estrategias e incluso puede ser definido con diferentes espacios de soluciones.

Bajo la denominación de **búsqueda exhaustiva** se engloban a aquellos problemas de fuerza bruta en los que se interna encontrar alguna solución en particular entre un conjunto de ellas de tipo combinatorio. Generalmente el espacio de búsqueda no es polinomial (aunque en situaciones podría serlo). Se cuenta con un conjunto de elementos entre los que elegir un subconjunto o un ordenamiento o una combinación entre ambos.

La metodología "**Generar y probar**" corresponde a tal vez la más sencilla para resolver un problema de búsqueda exhaustiva. La idea es simple: construir una función generativa que nos permitirá recorrer todo el espacio de soluciones posibles. El espacio de soluciones posibles estará restringido por la naturaleza del problema. Estas restricciones se conocen como **restricciones explícitas**. Por cada solución candidata se debe verificar su validez mediante una función de prueba. Esta función verifica las **restricciones implícitas** del problema. En caso de cumplir con los criterios de solución del problema, la misma es seleccionada. De lo contrario se procede a probar el siguiente candidato. De forma conceptual se lo puede resumir como:

Generar y probar: Esquema general

```
Sea S el dominio de soluciones posibles al problema P
Mientras queden soluciones candidatas por verificar y no se haya encontrado la solución
        Sea s la siguiente solución candidata a verificar
        Si s corresponde a una solución válida
                retornar s
```

Este esquema básico puede variar de acuerdo si lo que nos interesa es obtener todos los posibles resultados que cumplen un criterio, si solo nos interesa uno o si debemos comparar entre resultados válidos para quedarnos con el de mayor o menor valor según un criterio específico.

Cuando trabajos con espacios combinatorios es habitual que a medida que crezca el tamaño de la instancia se produzca un aumento importante del espacio de soluciones. Se conoce a este fenómeno como **explosión combinatoria**. Esta situación limita a estos métodos cuando la instancia del problema supera cierto tamaño. Por lo que su implementación se ve limitada.

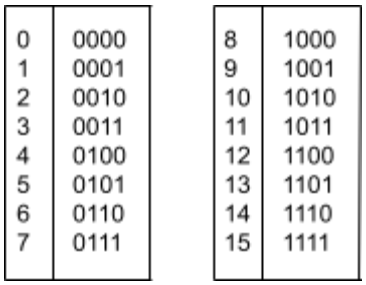
A continuación trabajaremos con algunos ejemplos de problemas que pueden ser resueltos mediante esta técnica y que presentan diferentes espacios de soluciones: n-tuplas. permutaciones, combinaciones y particiones de conjunto. Para cada caso se elaborará la estructura y el algoritmo para la generación de los candidatos de solución. Finalmente se trabajará con un caso diferente, búsqueda exhaustiva en grafos.

3. n-tuplas y el problema de la mochila

Supongamos que contamos con una mochila que puede contener hasta un límite de K kilos. Además podemos seleccionar entre “n” elementos para introducir en ella. Para cada elemento i se cuenta con dos valores asociados. Un valor v_i que representa cuánto ganamos si lo introducimos en la mochila y un peso k_i que representa cuánto espacio de la mochila necesita para poder incluirse. El objetivo que buscamos es seleccionar algunos de los elementos de tal forma de - sin superar la capacidad de la mochila - conseguir el mayor valor posible en ella. Formalizaremos el problema:

Problema de la Mochila
Contamos con una mochila con una capacidad de K kilos y queremos introducir dentro de ellas un subconjunto del conjunto E de “n” elementos con el objetivo de maximizar la ganancia. Cada elemento i tiene un peso de k_i kilos y un valor de v_i. Los elementos no pueden ser divididos y el peso total seleccionado no puede superar la capacidad de la mochila.

Podemos como estrategia de resolución ir seleccionando diferentes subconjuntos de E, probar si entran a la mochila y en caso afirmativo sumar su valor para obtener el valor del conjunto. Independientemente del resto, cada elemento disponible puede o no estar dentro de la selección. Las diferentes alternativas de elección de cada uno de los elementos conforman las soluciones posibles. Podemos expresar una posible configuración como un vector de n posiciones. Donde la posición i en el vector corresponde a la elección o no de ese elemento. Utilizaremos el valor 1 si el elemento se encuentra seleccionado y 0 en caso contrario.



0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Si tenemos 2 elementos entonces podremos tener 4 combinaciones posibles de elementos: ambos, ninguno o solo uno de los dos. Los podemos representar con un vector de 2 posiciones. Si tenemos 3 elementos, la cantidad posible de combinaciones corresponde a 8 y requerimos un vector de 3 posiciones. De igual forma, si tenemos 4 posibles elementos, la cantidad de soluciones posibles de combinaciones de elementos seleccionados corresponde a 16. Podemos representar al subconjunto de elementos seleccionados en un vector de n posiciones. En la imagen que acompaña este párrafo se muestran las 16 posibles combinaciones para seleccionar un total de 4 elementos.

Para el caso general con "n" elementos y utilizando un vector de "n" posiciones, podemos tener 2^n posible de combinaciones. Llamamos al proceso de construir estas posibles soluciones como la **generación de todas las n-tuplas**. Esto corresponde al espacio de las soluciones y define las restricciones explícitas de este problema. En este caso estamos generando cada uno de los elementos posibles en orden lexicográfico. Cada vector que representa una posible solución también representa en binario el orden en el que es encontrado y evaluado esa posible solución. Por lo tanto para la generación de estas soluciones podemos utilizar un contador binario.

El siguiente pseudocódigo representa una posible construcción de un contador binario:

Contador binario
Sea C un vector de n posiciones

```
Inicializar C:
        Establecer cada posición del vector la C[x] en 0.

Incrementar C:
        Sea pos=n-1
        Mientras C[pos]==1 y pos>0
                C[pos]=0
                Decrementar pos
        Si pos==-1
                Retornar 'fin'  //Overflow!
        C[pos]=1
        retornar C
```

Se proveen dos funciones que realizan el trabajo. El primero inicializa el vector en todos los elementos en 0. Para el problema de la mochila representa la posible solución de no incluir ningún elemento en la misma. Este proceso se realiza en O(n). El proceso de incremento obtiene el próximo subconjunto a probar. Se puede observar que invierte los valores en 1 de los elementos del vector comenzado por la derecha hasta encontrar un elemento en 0. Que lo invierte en un 1. En el peor de los casos se modifican "n" valores del vector. Sin embargo se puede ver que en promedio al realizar todo el proceso por cada operación de incrementar se terminan realizando un trabajo de complejidad O(1). Esta complejidad se puede demostrar mediante el análisis amortizado de esta estructura.

Por cada posible solución que se construye se debe verificar si corresponde a una que puede ingresar a la mochila (su peso combinado es igual o menor a la capacidad de la mochila. Esta verificación corresponde a las restricciones implícitas del problema. Si cumple con la condición diremos que corresponde con una **solución factible**. Esta definición se utiliza en muchos problemas de optimización: corresponde a una solución que cumple con todos los requisitos del problema, pero que no necesariamente satisface el criterio de optimizar (maximizar o minimizar) el resultado buscado.

Las siguientes funciones se pueden utilizar para evaluar las posibles soluciones:

Evaluación de posibles soluciones

```
Sea C un vector representando un subconjunto de elementos

Es Factible C:
```

```
        Sea pesoNecesario = 0
        Desde i=0 hasta n-1
                pesoNecesario += C[i] * k[i]
        retornar ( pesoNecesario <=K)

Ganancia C:
        Sea valorTotal = 0
        Desde i=0 hasta n-1
                valorTotal += C[i] * v[i]
        retornar valorTotal
```

Por un lado se evalúa si un subconjunto corresponde a una solución factible y en segundo lugar se calcula el valor de los elementos. Al ser una iteración sobre un vector de "n" elementos cada una de estas funciones realizan su trabajo con una complejidad de O(n).

Unificando cada una de las partes se puede construir el algoritmo de generar y probar para resolver el problema de la mochila:

Problema de la mochila: Solución mediante generar y probar

```
Sea C un vector representando un subconjunto de elementos
Inicializar C
Sea maximaGanancia = 0
Sea soluciónMáxima = C

Mientras Incrementar C <> 'fin'
        Si Es Factible C y Ganancia C > maximaGanancia
                maximaGanancia = Ganancia C
                soluciónMáxima = C

retornar soluciónMáxima y maximaGanancia
```

La complejidad de este algoritmo está dada por el iterador de soluciones que se ejecutará $O(2^n)$ veces y la complejidad del trabajo dentro del iterador que corresponde a O(n). Por lo tanto resolver el problema de la mochila mediante generar y probar lo estamos realizando en $O(n2^n)$.

Antes de finalizar cabe aclarar que no es la única manera de generar los diferentes subconjuntos. Una alternativa corresponde a construir las soluciones en el conocido **Gray binary code**. En esta metodología cada subconjunto construido se diferencia del anterior

en un solo elemento modificado. Es decir que uno de ellos pasa a estar o no comparándolo con el subconjunto anterior. Ciertamente la cantidad de soluciones posibles a probar no se modifica. En la tabla siguiente se muestran el orden de 16 números binarios de Gray. Sin embargo, la ventaja es que en muchas situaciones se puede calcular la nueva solución de forma más rápida. En el problema de la mochila solo se debe sumar o restar el peso y valor de un único elemento comparado con la solución del paso anterior. Logrando en última instancia disminuir levemente la complejidad final del algoritmo. Existen también diferentes maneras de implementar los códigos binarios de Gray. Un compendio de ellos se puede encontrar en la publicación "Combinatorial Gray codes-an updated survey"[1]

0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100

8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

4. Permutaciones y el problema del viajante

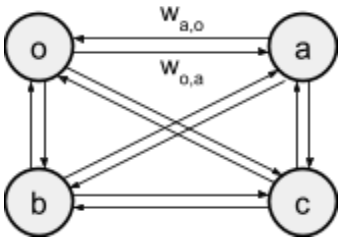
Supongamos que debemos visitar un conjunto de "n" ciudades del mapa partiendo de una cierta ciudad inicial. Supongamos que estas ciudades están intercomunicadas entre sí de modo que podemos ir de cualquiera de ellas a otra sin pasar por ninguna intermedia. El recorrido no debe repetir ciudades previamente visitadas y además debe finalizar en la misma ciudad desde donde se partió. Cada camino que une dos ciudades tiene asociado un coste por transitarlo. Es posible que algunos pares de ciudades no tengan un camino que los unan entre sí. Los caminos son direccionados. El objetivo final del problema es encontrar el recorrido que cumpla con los requerimientos de menor costo total posible. En la literatura se conoce a este problema como "el del viajante de comercio" y se puede formalizar de la siguiente forma:

Problema del viajante de comercio
Contamos con un conjunto de "n" ciudades que visitar. Existen caminos que unen pares de ciudades. Cada camino inicia en una ciudad "x" y finaliza en la ciudad "y" tiene asociado un costo de tránsito de $w_{x,y}$. Partiendo desde una ciudad inicial y finalizando en la misma se

[1] Mutze, Torsten. "Combinatorial Gray codes-an updated survey." (2022).

quiere construir un circuito que visite cada ciudad una y solo una vez minimizando el costo total.

Supongamos que contamos con 3 ciudades {o, a, b}. Además que existen todos los caminos posibles entre cada una de las ciudades. La "o" corresponde a la ciudad inicial (o de origen). Se puede comprobar que solo hay 2 circuitos posibles [o,a,b,o] y [o,b,a,o]. El costo del primer camino corresponde a $w_{o,a}+w_{a,b}+w_{b,o}$ *y el del segundo camino* $w_{o,b}+w_{b,a}+w_{a,o}$*. Podemos obviar en la expresión del camino la ciudad inicial y final que siempre corresponde a la misma. Llamamos m con* $m=n-1$ *a la cantidad de ciudades en esta expresión. Si, ahora tenemos 4 ciudades la cantidad de posibles circuitos corresponde a 6: [a,b,c], [a,c,b], [b,a,c], [b,c,a], [c,a,b], [c,b,a]. En el grafo se muestran los posibles caminos a recorrer para construir el circuito que parte de la ciudad "o" y finaliza en la misma.*



En el caso general podemos determinar que con n ciudades (y m ciudades interiores en el circuito) tenemos m! posibles circuitos. Corresponde, si lo expresamos en función de n, a (n-1)!. Llamamos a la generación de todos los posibles circuitos o dominio de las soluciones a la generación de las **permutaciones de m elementos**. Conforman el espacio de soluciones y se ajustan a las restricciones explícitas del problema. Una posible permutación de m elementos lo podemos representar con un vector de m posición. En cada posición se almacena el elemento en el orden a evaluar. Podemos generar las permutaciones en orden lexicográfico. Para eso iniciaremos con un vector inicial ordenado de menor a mayor. Posteriormente iremos rotando de a uno ciertas posiciones para lograr las m! permutaciones. La imagen que acompaña el párrafo contiene un ejemplo de las permutaciones ordenadas lexicográficamente de 4 elementos. En este orden se puede observar que se inicia con un vector ordenado y se finaliza con el orden inverso. Entre cada permutación y la siguiente se conserva un prefijo en el vector. Cuando el prefijo cambia corresponde a la inexistencia de posibles soluciones adicionales que comiencen con ese mismo orden.

0	abcd
1	abdc
2	acbd
3	acdb
4	adbc
5	adcb
6	bacd
7	badc

8	bcad
9	bcda
10	bdac
11	bdca
12	cabd
13	cadb
14	cbad
15	cbda

16	cdab
17	cdba
18	dabc
19	dacb
20	dbac
21	dbca
22	dcab
23	dcba

El siguiente algoritmo es acreditado por Donald Knuth[2] al matemático del siglo 14 del norte de la India, Narayana Pandita. Genera las permutaciones en el orden lexicográfico. Se puede dividir en dos procedimientos. El primero de inicialización donde se construye el vector con los elementos ordenados de menor a mayor. El segundo, un proceso que genera la siguiente permutación en base a la última generada. Este último se puede dividir en 3 partes. En la primera, se busca el prefijo más grande al que se le han elaborado todas las permutaciones que empiezan con el mismo. Para encontrarlo se aprovecha del orden lexicográfico. El siguiente paso corresponde a reemplazar el último elemento del prefijo con el elemento más pequeño a su izquierda que sea superior a este. Finalmente, como último paso se ordena de menor a mayor los elementos por fuera del prefijo. La generación de permutaciones finaliza cuando los elementos quedan ordenados de mayor a menor (equivalente a la ausencia de prefijo).

Un posible pseudocódigo para el método es el siguiente:

Generador de permutaciones

```
Sea C un vector representando un orden de visita de las m ciudades internas

Inicializar C:
        Establecer en cada posición C[i] el elemento i ordenado lexicográficamente.

permutar C:
        Sea indice1=m-1
        Mientras C[indice1]>=C[indice1+1]
                indice1 -= 1

        Si indice1 == 0
                retornar 'fin'

        Sea indice2=m
        Mientras C[indice1]>=C[indice2]
                indice2 -= 1

        Intercambiar C[indice1] y C[indice2]
```

[2] Donald E. Knuth. The Art of Computer Programming: Combinatorial Algorithms, Part 1 (1st. ed.). Addison-Wesley Professional. 2011.

```
    Sea k = indice1 + 1
    Sea l = n

    Mientras k < l
        Intercambiar C[k] y C[l]
        k += 1
        l -= 1

    retornar C
```

Supongamos que tenemos 4 ciudades (más la ciudad inicial) y queremos construir todas las permutaciones de ellas posibles. En primer lugar al inicializar, construimos el vector con el siguiente ordenamiento a,b,c,d. Será el primero al que evaluar su costo. Para generar la siguiente permutación se debe invocar a "permutar C". La primera vez que es llamado, solo se ha visitado la solución a,b,c,d y el prefijo a,b,c es aquel en el que ya se han generado todas las permutaciones que inician con este. Siguiendo el pseudocódigo se observa que se comienza la búsqueda en la anteúltima posición del vector C[m-1]=c y se compara con C[m]=d. Como "c" es menor que "d" no ingresa al iterador. indice1 queda con valor m-1. Note se que en el caso donde el vector esté ordenado en forma invertida (en este caso d,c,b,a) finaliza la ejecución dado que ya se han generado todas las permutaciones posibles. Esto ocurre si indice1 queda en la posición 0. En el paso siguiente busca utilizando la variable indice2 al primer elemento iterando de izquierda a derecha que sea mayor al elemento en indice1. Al encontrarlo lo reemplaza. Finalmente ordena en la tercera iteración los elementos entre indice1+1 al final de menor a mayor. El resultado corresponde a la permutación a,b,d,c.

Si bien en el peor de los casos se recorren todos los elementos del vector en un llamado de la función permutar, se puede calcular en forma amortizada la complejidad de ejecución en O(1).

Una vez que contamos con una permutación debemos comprobar si el camino es posible (todas las ciudades contiguas tienen un camino que las una) Si la respuesta es afirmativa, nos interesa el costo de realizar el circuito en este orden. Planteamos utilizar como indicador de la inexistencia de un camino entre dos ciudades el valor infinito. Asi si existe un camino posible siempre será menor a esté y lo seleccionará. El siguiente pseudocódigo realiza la evaluación del costo del circuito:

Evaluación de posibles soluciones

```
Sea C un vector representando un orden de visita de las m ciudades internas
Sea o la ciudad de inicio y finalización
Sea w[x,y] el costo de transitar por el camino que une la ciudad x a la y.

Costo C:
        Sea costoTotal = w[o,C[1]]
        Desde i=1 hasta m
                costoTotal += w[C[i],C[i+1]]
        costoTotal += w[C[m],o]

        retornar costoTotal
```

El costo total se calcula evaluando la suma de los costos de los caminos que lo integran. Al ser una iteración sobre un vector de "n-1" posiciones realiza su trabajo con una complejidad de O(n).

Unificando cada una de las partes se puede construir el algoritmo de generar y probar para resolver el problema del viajante de comercio:

Problema del viajante de comercio: Solución mediante generar y probar

```
Sea C un vector representando un orden de visita de las m ciudades internas
Sea o la ciudad de inicio y finalización
Sea w[x,y] el costo de transitar por el camino que une la ciudad x a la y.

Inicializar C

Sea minimoCosto = Costo C
Sea soluciónинima = C

Mientras permutar C <> 'fin'
        Si minimoCosto  > Costo C
                minimoCosto = Costo C
                soluciónинima = C

retornar soluciónинima y minimoCosto
```

La complejidad de este algoritmo está dada por el iterador de soluciones que se ejecutará O((n-1)!) veces y la complejidad del trabajo dentro del iterador que corresponde a O(n). Por lo tanto resolver el problema de la mochila mediante generar y probar lo estamos realizando en O(n!).

¿Qué pasaría si algunas de las ciudades están incomunicadas entre sí?. Para evitar modificar la solución se puede plantear en el grafo un costo de infinito entre ellas. de esa forma si existe un circuito de un valor menor a infinito se preferiría. Por lo tanto estamos forzando a no ir por un tramo que no existe. Por otro lado, si el resultado obtenido es infinito, nos informa que es imposible resolver el problema para la situación planteada.

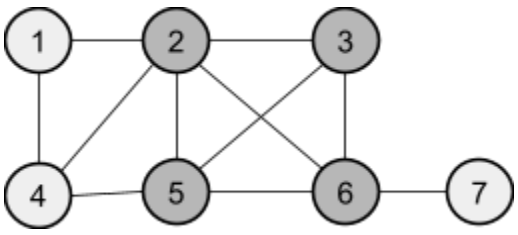
De igual manera que al generar n-tuplas, la generación de permutaciones puede realizarse en diferentes órdenes. Particularmente también, solo realizando el intercambio de dos elementos entre cada permutación. Ejemplo de ellos es el algoritmo de Heap[3] que según Robert Sedgewick[4] corresponde al más eficiente en cuanto a tiempo de generación. En la tabla se puede ver las permutaciones de 4 elementos generadas por este algoritmo. Una ventaja adicional al intercambiar solo un elemento es que en muchas ocasiones se puede utilizar la solución anterior para calcular la siguiente con menor esfuerzo. En nuestro ejemplo del problema del viajante intercambiar el orden de dos ciudades implica que se puedan modificar como mucho 4 caminos del circuito total. Para valores de "n" grandes, evitar el recálculo completo del costo hace una diferencia en el tiempo total de ejecución.

0	abcd
1	badc
2	cabd
3	acbd
4	bcad
5	cbad
6	dbac
7	bdac

8	adbc
9	dabc
10	badc
11	abdc
12	acdb
13	cadb
14	dacb
15	adcb

16	cdab
17	dcab
18	dcba
19	cdba
20	bdca
21	dbca
22	cbda
23	bcda

5. Combinaciones y el problema del clique



En un grafo no dirigido se conoce como **clique** a un subconjunto de vértices, en el que para cualquier par de ellos existe un eje que los une. Es decir que todos los vértices son adyacentes entre sí. Si

[3] B. R. Heap, Permutations by Interchanges, The Computer Journal, Volume 6, Issue 3, November 1963, Pages 293–298,
[4] Robert Sedgewick. 1977. Permutation Generation Methods. ACM Comput. Surv. 9, 2 (June 1977), 137–164.

consideramos de forma independiente a los vértices del clique vemos que conforman entre sí un grafo completo. Por ejemplo en el grafo de la imagen contamos con 7 vértices y podemos ver la existencia de un clique de tamaño 4 con los vértices 2,3,5 y 6.

Clique en inglés significa "pandilla" que corresponde a un conjunto pequeño de personas muy unido entre sí y cerradas al ingreso de otras. El concepto de cliques en grafos es utilizado en diversos escenarios. Por ejemplo buscar comunidades en redes sociales o en ciertas aplicaciones en química computacional.

Formalizaremos el problema de la siguiente manera:

Problema del clique
Dado un grafo no direccionado G=(V,E) con V su conjunto de vértices y E el de sus aristas. Queremos obtener, si existe, un clique de tamaño k dentro de él.

Si queremos encontrar un clique de tamaño c debemos encontrar esa misma cantidad de nodos que se comuniquen cada uno entre sí. Para realizarlo por fuerza bruta podemos probar toda combinación posible de "k" nodos del grafo. Diremos que la cantidad de nodos del grafo es igual a "n". Por lo tanto, la cantidad de casos a probar está dada por la cantidad de **combinaciones de "n" nodos tomados de a "k"**. Lo que se expresa como $\binom{n}{k}$ y se puede calcular como $\left(\frac{n!}{k!\,(n-k)!}\right)$.

El grafo presentado anteriormente tiene 7 nodos y buscamos un clique de tamaño 4. La cantidad de combinaciones de 7 nodos tomados de a 4 a evaluar se expresa como $\binom{7}{4}$ *y se puede calcular como* $\left(\frac{7!}{4!\,(7-4)!}\right) = 35$.

Antes de determinar cómo generar cada una de las combinaciones, es preciso elaborar una manera de representarlas. Una opción podría involucrar un vector binario de "n" posiciones. Donde una posible solución a verificar corresponda a "k" de sus posiciones con valor "1" y el resto con valor "0". Otra alternativa consiste en elaborar una lista de "k" elementos, donde en cada posición se ubica el identificador del elemento seleccionado. Utilizaremos esta última dado que se adecuan mejor a los procedimientos que realizaremos. Para representar los conjuntos lo realizaremos de forma lexicográfica.

Existen dos ordenamientos diferentes que cumplen con este requerimiento. Según si ordenamos los elementos en el vector de menor o mayor o de forma inversa. En este caso trabajaremos con la última.

En el ejemplo del grafo tendremos las 35 combinaciones mostradas en la tabla adjunta. Cada conjunto de nodos se encuentra ordenado de mayor a menor. Se puede verificar que el ordenamiento entre subconjuntos de vértices es lexicográfico.

0	4321
1	5321
2	5421
3	5431
4	5432
5	6321
6	6421
7	6431

8	6432
9	6521
10	6531
11	6532
12	6541
13	6542
14	6543
15	7321

16	7421
17	7431
18	7432
19	7521
20	7531
21	7532
22	7541
23	7542

24	7543
25	7621
26	7631
27	7632
28	7641
29	7642
30	7643
31	7651

32	7652
33	7653
34	7654

Con el objetivo de crear algorítmicamente las combinaciones propondremos utilizar un vector C de k+2 posiciones. Las primeras dos se utilizaran como "centinelas" es decir como variables de control para determinar cuándo finalizar el procesamiento y cuándo aumentar la última cifra. Además por conveniencia expresaremos los índices del vector de forma inversa: C[k+2],C[k+1],C[k],...,C[1]. Utilizaremos un proceso de inicialización en el que cargaremos la primera combinación. Luego en un proceso iterativo se irán construyendo los diferentes subconjuntos. Solo utilizaremos del vector las posiciones de 1 a k para la evaluación de la combinación construida. El "incrementar" comienza por la posición 0 y busca el primer lugar donde los elementos contiguos no son inmediatamente contiguos. Cuando eso ocurre incrementa en uno el elemento donde se encontró la diferencia. Además todos los elementos anteriores los pasa a los valores de inicialización.

Generador de combinaciones

Sea C un vector representando el conjunto de vértices a evaluar con dos posiciones adicionales.

```
Inicializar C:
        Desde i=1 a k
                Establecer C[i]=i
        C[k+1]=n+1
        C[k+2]=0

Incrementar C:
        Sea j=1
        Mientras C[j]+1==C[j+1]
```

```
            C[j] =  j
            j+=1
    Si j>k
            retornar 'fin'

    C[j]+=1
    retornar C
```

Para el problema del grafo G con 7 nodos y donde buscamos un clique de tamaño 4, construiremos entonces un vector con 6 posiciones. 4 para la combinación y los restantes como centinelas. Al inicializar nos quedará C=[0,8,4,3,2,1]. El primer candidato a evaluar corresponde a [4,3,2,1]. Al llamar por primera vez al incrementar se puede observar que hasta la posición 4 todos los valores son contiguos entre sí (C[4]=4 y c[5]=8). Por lo que se incrementa c[4] en 1 y los anteriores se establecen en su valores de inicialización. El vector queda como C=[0,8,5,3,2,1] (y la combinación como 5,3,2,1). El proceso se vuelve a repetir. El primer elemento que no es contiguo corresponde a C[3] que se incrementa en 1. El vector quedará representado como C=[0,8,5,4,2,1] (y la combinación como 5,4,2,1). Se realizará un proceso iterativo hasta que el vector quede como C=[0,8,7,6,5,4]. En este caso el primer elemento no contiguo corresponde a C[5]. Esta posición del vector está por fuera de la representación de la combinación. Por lo tanto, corresponde a un indicador de la finalización del procesamiento. Ya se generaron todas las combinaciones posibles.

Este algoritmo es presentado por Knuth en "The Art of Computer Programming: Combinatorial Algorithms, Part 1" y llamado como algoritmo "L". La complejidad amortizada de cada incremento es O(k) en el peor caso. Por lo tanto la complejidad total del algoritmo es $O\left(k\frac{n!}{k!\,(n-k)!}\right)$. Existen otros algoritmos de generación más eficientes (y conceptualmente algo más complejos). Varios de ellos presentados en el mismo libro de Knuth.

Una vez que tenemos una combinación debemos verificar si forma un clique. El siguiente pseudocódigo realiza el trabajo.

Evaluación de posibles soluciones

```
Sea C un vector con los nodos a evaluar
Sea M la matriz de adyacencia del grafo G

Es_clique C:
```

```
Desde elemento1=1 a k
        Desde elemento2=elemento1+1 a k
                Si M[C[elemento1],C[elemento2]]==0
                        Retornar 'No'
Retornar 'Si'
```

Si el grafo se tiene almacenado como una matriz de adyacencia, consultar si un par de ejes están conectados mediante un eje corresponde a una consulta O(1). Por lo tanto si contamos con k nodos y queremos determinar si corresponde a un clique debemos por cada uno de ellos verificar si está conectado con los otros k-1. Como hablamos de un grafo no dirigido, podemos evitar verificar la existencia de un eje en cada uno de los sentidos. El pseudocódigo presentado realiza la verificación en $O(k^2)$.

Unificando cada una de las partes se puede construir el algoritmo de generar y probar para resolver el problema del clique:

Problema del clique: Solución mediante generar y probar

```
Sea C un vector con los nodos a evaluar
Sea M la matriz de adyacencia del grafo G

Inicializar C

Repetir
        Si Es_clique C
                retornar C
Hasta que Incrementar C == 'fin'

retornar 'No hay solución'
```

La complejidad total de este algoritmo corresponde a $O\left(k^2 \frac{n!}{k!\,(n-k)!}\right)$.

6. Particiones de conjunto y el problema del clustering

Supongamos que contamos con "n" elementos (u objetos) con ciertas características y queremos agruparlos según sean similares entre sí. Para determinar la similaridad entre elementos pueden existir diferentes métodos. No es relevante para nuestra presentación del problema cuál de ellos se seleccione. Llamamos a una agrupación de varios elementos

un **cluster**. Así como se establece una medida de similitud entre elementos, se suele definir una medida similar inter cluster. Se espera que los elementos de un mismo cluster sean los más parecidos entre sí y los que están en otros sean lo más diferente posible. Estas dos medidas resultan en ocasiones antagónicas entre sí. Por lo que también se suele asignar alguna relación entre ambas. La elección de diferentes funciones de medida y su relación determinan diferentes agrupaciones resultantes. A priori no conocemos cuántos cluster se obtendrán, por lo tanto para obtener una solución óptima se deben evaluar todas las posibles particiones de los elementos.

Formalizaremos el problema de la siguiente manera:

Problema del clustering
Dados "n" elementos con ciertas características mensurables y comparables queremos agruparlos en clusters de forma de lograr un balance que minimice la distancia entre los puntos dentro de un mismo cluster y maximizar la distancia entre los puntos entre clusters diferentes.

Una solución por fuerza bruta de "generar y probar" debe considerar todos los clusters posibles a formar sin considerar a priori que algunos sean mejores que otros. Para lograrlo deberemos tomar los n elementos como un conjunto inicial e ir particionándolos en diferentes subconjuntos. Llamaremos a este proceso la partición del conjunto.

Las **particiones de un conjunto** son las formas de representar ese conjunto como una unión de subconjuntos disjuntos no vacíos llamados bloques. Con el objetivo de estandarizar la representación mostraremos los subconjuntos como una lista de sus elementos ordenados lexicográficamente separados por el símbolo pleca (Barra vertical). Los subconjuntos también los ordenaremos entre sí de forma lexicográfica por su primer elemento.

0	1234
1	123 | 4
2	124 | 3
3	12 | 34
4	12 | 3 | 4
5	134 | 2
6	13 | 24
7	13 | 2 | 4

8	14 | 23
9	1 | 234
10	1 | 23 | 4
11	14 | 2 | 3
12	1 | 24 | 3
13	1 | 2 | 34
14	1 | 2 | 3 | 4

Si el conjunto está formado por 4 elementos {1,2,3,4}, las 15 diferentes formas de particionarlos son las mostradas en la tabla adjunta. Si tomamos la representación 14 \| 23 representa a los subconjuntos {1,4} y {2,3}. Este último es equivalente al {3,2} y

{4,1}. No nos interesa ni el orden de los elementos dentro de cada subconjunto, ni el orden de los subconjuntos.

La cantidad de particiones de conjunto posibles aumenta a medida que la cantidad de elementos del conjunto lo hace. Un solo elemento (n=1) nos permite una única partición. Dos elementos, esa misma cantidad de particiones. Al llegar a 3 elementos, tendremos 5 y ya vimos que con 4, 15. Peirce[5] pudo calcular la cantidad de particiones en función de la cantidad de elementos utilizando un triángulo de números.

1					
2	1				
5	3	2			
15	10	7	5		
52	37	27	20	15	
203	151	114	87	67	52

La primera columna (y también la diagonal) representan la cantidad de particiones existentes para la cantidad de elementos "n" (donde n es la fila del triángulo). Llamaremos $b_{x,y}$ a cada uno de los elementos del triángulo. El cálculo de cada elemento se realiza sumando su elemento siguiente en la fila y el superior en su columna, conformando una recurrencia. Inicialmente $b_{1,1}=1$. Luego para cada fila tendremos $b_{n,n}= b_{n-1,1}$. si $n>1$ y $b_{n,k}= b_{n-1,k} + b_{n-,k+1}$ para $1\leq k < n$. Se conoce a la secuencia obtenida por la primera fila $b_{1,n}$ o más generalmente como B_n, como números de Bell (En honor a E. T. Bell que realizó trabajos influyentes con estos). A medida que crece "n" el crecimiento de B_n se puede acotar por $\Theta\left(\frac{n}{ln\, n}\right)^n$ que nos brinda un estimado de la cantidad de particiones de conjunto que deberemos generar para cubrir todas las posibilidades para resolver nuestro problema.

Antes de presentar el algoritmo, estableceremos una manera de representar la partición mediante un vector P de "n" posiciones. Cada posición corresponde a un elemento y su valor numérico determina en qué conjunto se encuentra en la partición. Elementos con igual valor equivalen a elementos en el mismo subconjunto. Siempre el primer elemento se asignará a la partición 1. El resto de los elementos puede ser asignado a ese u otro, hasta un total de "n" distintos. Un elemento del vector en la posición j debe ser menor o igual al máximo valor de los elementos anteriores más uno. Este tipo de representación se conoce como cadena de crecimiento restringido (restricted growth string).

Regresamos al ejemplo formado por 4 elementos {1,2,3,4}. La siguiente corresponde a las 15 particiones posibles expresadas como cadena de crecimiento restringido: 1111, 1112, 1121,

[5] Peirce, C.S. American Journal of Mathematics, 3, 1880, p 15-57.

1122, 1123, 1211, 1212, 1213, 1221, 1222, 1223, 1231, 1232, 1233, 1234. En esta representación se cumple la condición de crecimiento restringido y el resultado corresponde a una secuencia ordenada lexicográficamente.

Ahora sí, es momento de desarrollar el algoritmo que construirá las particiones. Corresponde al algoritmo presentado por George Hutchinson[6]. Utilizaremos además de P, a un vector auxiliar A. Donde A[j] = max(P[i]) con 1<i<j-1. Es decir el valor máximo que aparece en alguna posición anterior. Además una variable r que contiene el valor más grande en A (que por la forma que se construye A corresponde a A[n]).

Inicialmente se establece que P=[1,1,...,1] correspondiendo a la primera partición posible donde todos los elementos se encuentran en el mismo conjunto. El vector auxiliar A tendrá los mismos valores siguiendo la regla de asignación del mismo. r tendrá como valor también 1. A partir de esta configuración se irán construyendo las particiones posibles.

Comenzando por P[n] se recorre el vector en forma decreciente hasta encontrar la primera posición "incrementable". Una posición j de P es incrementable si P[j] ≤ max(P[i]) con 1<i<j-1. Lo que es equivalente a compararlo a A[i]. El algoritmo incrementa esta posición en 1 a la vez que regresa a todas las posiciones anteriores al valor 1 y actualiza el vector auxiliar A para reflejar los nuevos máximos. El algoritmo finaliza cuando no se puede encontrar una posición incrementable. Esto ocurre cuando P = [1,2,...,n].

Generador de particiones del conjunto

```
Sea P un vector representando la partición en subconjuntos
Sea A un vector auxiliar
Sea r una variable auxiliar

Inicializar P:
        Desde i=1 a n
                Establecer P[i]=1
                Establecer A[i]=1
        r = A[n]

Incrementar P:
        Sea j=n
```

[6] George Hutchinson. Partitioning Algorithms for Finite Sets. Commun. ACM 6, 10 (Oct. 1963), p613–614.

```
Mientras P[j]>A[j]
        j-=1
Si j==1
        retornar 'fin'

P[j]+=1
r = max(P[j],A[j])

j += 1

Mientras j<=n
        Establecer P[j] = 1
        Establecer A[j] = r
        j +=1

retornar P
```

Para el problema donde contamos con n=4 elementos y queremos generar sus diferentes particiones, comenzaremos con el vector P=[1,1,1,1] y el vector auxiliar A=[1,1,1,1] y r=1. En el primer incremento buscaremos el primer elemento de P comenzando desde "n" que sea incrementable. Vemos que P[n=4] no es mayor a A[4], por lo tanto incrementamos a P[4]. La nueva configuración será P=[1,1,1,2] y el vector auxiliar A=[1,1,1,1]. En el segundo incremento observamos que P[4] es mayor a A[4]. Repetimos la comparación para P[3] y A[3]. En este caso se verifica la comparación. Por lo tanto se incrementa P[3] y a se establece P[4]=1. Además se actualiza el vector A. Quedan ambos vectores de la siguiente manera: P=[1,1,2,1] y el vector auxiliar A=[1,1,1,2]. En la tercera iteración P[4] es incrementable, quedando entonces P=[1,1,2,2] y sin modificarse A. En la cuarta iteración P[4] es incrementable, generando como resultado P=[1,1,2,3] y nuevamente si realiza el cambio de A. El proceso continuará hasta que llegue a la configuración P=[1,2,3,4] y A=[1,1,2,3]. En ese caso el elemento incrementable corresponde a P[j=1] y corresponde entonces a la finalización de procesamiento.

La complejidad total de este algoritmo corresponde a $O\left(\left(\frac{n}{ln\, n}\right)^n\right)$. Se puede verificar que el la inspección y modificación de los vectores en el "incremento" es O(1) de forma amortizada.

Supongamos que contamos con las funciones para evaluar la "calidad" de los clusters formados y su complejidad. Los llamaremos respectivamente "Evaluar P" y $f_e(P)$. Podemos expresar la resolución del problema del clustering utilizando generar y probar como;

Problema del clustering: Solución mediante generar y probar

```
Sea P un vector representando la partición en subconjuntos
Sea A un vector auxiliar
Sea r una variable auxiliar

Inicializar P

Sea valorSolucion = Evaluar P
Sea mejorSolucion = P

Mientras permutar C <> 'fin'
        Si valorSolucion  > Evaluar P
                minimoCosto = Evaluar P
                mejorSolucion = P

retornar mejorSolucion y valorSolucion
```

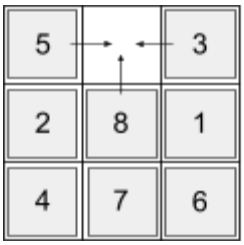
La complejidad total de este algoritmo corresponde a $O\left(\left(\frac{n}{ln\, n}\right)^n f_e(P)\right)$.

7. Espacio de estados finitos y recorrido exhaustivo de un grafo

Muchos procedimientos para la resolución de problemas se pueden pensar como una serie de decisiones que se realizan sobre la instancia del mismo que modifican su estado. El encadenamiento de esas decisiones parten de un estado inicial y establecen estados intermedios con el objetivo de llegar a una situación final que llamamos solución. La cantidad de estados posibles de un problema depende de la naturaleza del mismo. Las decisiones sobre un estado corresponden a las acciones que realizamos sobre estos que los transforman a un estado diferente.

Pensemos en algunos de los problemas presentados en secciones anteriores. El problema de la mochila puede visualizarse como un estado inicial que corresponde a una mochila vacía y como

estados intermedios a aquellos que corresponde a las diferentes configuraciones de elementos que podemos tener dentro de la misma. Una transición entre estados corresponde a agregar o quitar por ejemplo un elemento. En cada estado tenemos un peso dentro de la mochila y una ganancia posible. Uno (o un subconjunto) de todas los estados corresponde a la selección de elementos que maximizan la ganancia posible (sin superar la capacidad de la mochila) y ese será el estado final buscado. En el problema del viajante de comercio el estado inicial se puede pensar como un cierto camino que sale de la ciudad de inicio y recorre todas para regresar al origen. Luego cada decisión corresponde a cambiar un par de ciudades en el recorrido. Un nuevo estado corresponde al nuevo circuito resultante.

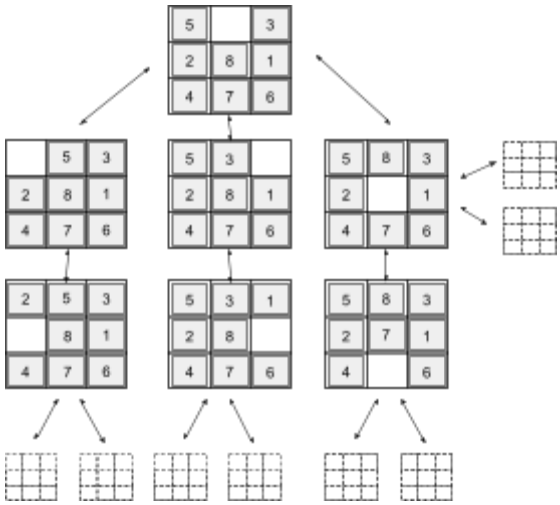


Propondremos como ejemplo de trabajo el problema conocido como 8-puzle. Contamos con un tablero de 3 filas por 3 columnas y un conjunto de 8 piezas deslizantes numeradas del 1 al 8. Las piezas pueden estar ordenadas en cualquier posición inicial. Las piezas se pueden mover de forma horizontal (izquierda o derecha) y vertical (arriba o abajo) siempre que el destino corresponda a la única posición vacía del tablero. El objetivo intentar (si se puede) ordenar las piezas de forma que se puedan leer de corrido de forma ascendente sus números y que el espacio vacío quede en el extremo inferior derecho. En la imagen se muestra una posible configuración inicial. Sus posibles transiciones a nuevos estados están representados por el movimiento de alguna de sus piezas. A la derecha se observa la posición final buscada.

Podemos representar esta situación utilizando un grafo que llamaremos **grafo de espacio de estados**. Un nodo de este grafo corresponde a un estado del problema. Un eje, corresponde a una acción que permite transicionar de un estado a otro. En ocasiones estos ejes son dirigidos y en otros no. Usualmente un mismo problema permite ser pensado y representado de formas diferentes. Lo que ocasiona que el tamaño del grafo de espacio de estados sea diferente.

Regresando al problema de 8-puzzle podemos pensar al grafo donde tendremos dos nodos que corresponden al estado inicial y al final. Además tendremos un nodo intermedio por cada posible estado posible que corresponde a los diferentes ordenamientos de las piezas deslizantes. Algunos de estos nodos están conectados entre sí mediante un eje bidireccional si el

deslizamiento de alguna de sus piezas permite transicionar desde un estado al otro. La imagen que acompaña este párrafo contiene una porción del grafo para el ejemplo presentado anteriormente. Si analizamos el problema en profundidad llegaremos a la existencia de 9! estados posibles. Sin embargo se puede demostrar que no todos los estados son alcanzables dada una configuración inicial (de hecho, sólo la mitad de los 9!).



Una vez construido el grafo o mediante algún algoritmo que lo permita construir dinámicamente, se debe recorrer partiendo de la solución inicial para hallar la solución final. Existen diferentes maneras de recorrer este árbol. Veremos a continuación dos maneras de hacerlo: Búsqueda en profundidad (Depth-first Search) y búsqueda en anchura (Breadth-first Search). Estos métodos son clásicos y son utilizados para una gran variedad de situaciones.

La **búsqueda por profundidad** o DFS (por sus siglas en inglés) fue propuesto por Charles Pierre Trémaux (1859–1882) como estrategia para resolver laberintos. Se puede resumir el procedimiento en iniciar el recorrido de una posición inicial (estado inicial o nodo de partida). Determinar cuales son las próximas posiciones (estados a transicionar o nodos adyacentes) a las que se puede transitar. Seleccionar a una de ellas previamente no visitadas o retroceder a un paso anterior si no hay donde ir. Se debe marcar los lugares visitados. Luego repetir el procedimiento hasta encontrar la solución (estado final o nodo de llegada).

La implementación presentada utiliza una estructura de datos tipo pila para determinar los caminos descubiertos y por evaluar. El último último "apilado" será el primero en visualizar. El resto, se explorarán cuando no sea posible agregar nuevos caminos y se deba retroceder a uno de los caminos previamente encontrados y no explorados. Es fundamental ir marcando aquellos nodos ya analizados y de esa manera evitar volver a transitarlos. El pseudocódigo corresponde a:

Algoritmo de Búsqueda por profundidad iterativo (DFS)

```
Sea G=(V,E) un grafo de |V|=n nodos y |E|=m ejes
Sea o un nodo de G inicial.
Sea P una pila de nodos

Agregar o a la pila
Mientras la pila tenga nodos
        Obtener nodo v de la pila
        Si v aún no fue visitado
                marcar v como visitado
                Para todo nodo u adyacente a v
                        Agregar u a la pila
```

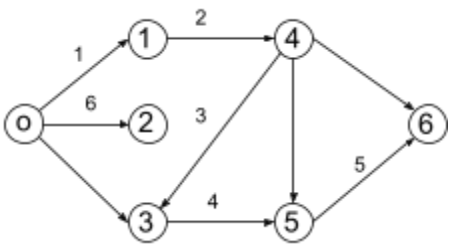
En su versión recursiva se puede expresar como:

Algoritmo de Búsqueda por profundidad recursivo (DFS)

```
Sea G=(V,E) un grafo de |V|=n nodos y |E|=m ejes
Sea o un nodo de G inicial.

DFS (nodo u):
        Establecer u como visitado
        Para todo nodo v adyacente a u
                sI v no fue visitado
                        DFS(v)
```

Ambos métodos se presentan en forma esquemática. Dependiendo del problema a resolver se deben agregar los procedimientos necesarios. Por ejemplo, determinar el camino realizado para llegar a este punto o verificar si la nodo corresponde a la solución final (o si es un problema de optimización corresponde a una solución mejor a las obtenidas previamente).

La complejidad temporal de ambas presentaciones es en función de la cantidad de nodos y ejes del grafo. En primer lugar se visitará una única vez cada nodo. Cuando se agrega a la pila o se llama recursivamente se analizan todos los nodos adyacentes del nodo actual. Por lo tanto al finalizar se habrán analizado una vez cada uno de los ejes del grafo. La utilización de una estructura para marcar los ejes y el uso de una lista de adyacencias logra que la complejidad final sea acotada por O(n+m).



Supongamos que el grafo a recorrer corresponde al de la figura. Iniciamos en el nodo "o". Sus nodos adyacentes corresponden a los nodos 1,2 y 3. Se seleccionará uno de ellos, por ejemplo el 1. Se marca como visitado. Los nodos 2 y 3 quedan pendientes de visita. Luego desde el nodo 1 se seleccionará el nodo 4 que también se marcará como visitado. Desde este último se puede visitar el 3 y el 6. Se selecciona el 3 y se marca. Desde allí al 5 y luego al 6. Como en este último no hay caminos nodos por visitar, se retrocede al 5. Ocurre lo mismo y se regresa al 4. Se verifica que tanto los nodos 3,5 y 8 ya fueron visitados, nuevamente se retrocede. Lo mismo ocurre y estamos nuevamente en el nodo inicial. Desde allí se observa que el nodo 2 aun no se visito, por lo tanto se realiza, para luego finalmente retroceder. No quedan nodos por visitar, por lo tanto el proceso finaliza. En la imagen se muestra sobre los ejes el orden de visita de los mismos.

La **búsqueda por anchura** o BFS (por sus siglas en inglés) propuesto por diversos autores entre ellos Konrad Zuse y Michael Burke (1945), Edward F. Moore (1959) y C. Y. Lee (196). Al igual que DFS también inicia el recorrido de una posición inicial. A partir de este obtiene todos los nodos adyacentes a este que aun no se hayan visitado, los agrega a una cola que indica el orden de visita. Luego por cada elemento en la cola lo visita y obtiene también sus adyacentes, agregándolos al final de la cola. Repite el procedimiento hasta que no haya elementos en la cola. Su funcionamiento se puede resumir como "el primero encontrado será procesado antes". El pseudocódigo corresponde a:

Algoritmo de Búsqueda por anchura iterativo (BFS)

```
Sea G=(V,E) un grafo de |V|=n nodos y |E|=m ejes
Sea o un nodo de G inicial.
Sea P una lista de nodos

Agregar o a la cola
Mientras la cola tenga nodos
        Obtener nodo v de la cola
        Si v aun no fue visitado
                marcar v como visitado
                Para todo nodo u adyacente a v
```

```
                Agregar u a la cola
```

En su versión recursiva se puede expresar como:

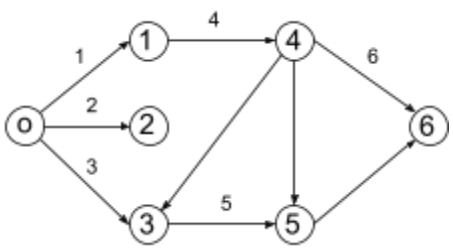
Algoritmo de Búsqueda por anchura recursivo (BFS)

```
Sea G=(V,E) un grafo de |V|=n nodos y |E|=m ejes
Sea o un nodo de G inicial.
Sea l una cola de nodos
Encolar o en l.

BFS (l):
        Obtener nodo u de la cola l
        sI u no fue visitado
                Establecer u como visitado
                Para todo nodo v adyacente a u
                sI v no fue visitado
                        Agregar v a la cola l

        si hay elementos en la lista l
                BFS(l)
```

Al igual que el DFS presentado, dependiendo del problema a resolver se deben agregar los procedimientos necesarios. Una ventaja de BFS es que si estoy buscando un camino de un nodo inicial a uno final, la forma de recorrer por el algoritmo nos asegura encontrar un camino de longitud mínima. Se puede ver que los nodos en el grafo se van recorriendo "en capas". La primera capa corresponde a los nodos adyacentes al nodo inicial. La segunda a aquellos nodos accesibles desde un nodo de la primera capa y sucesivamente.

La complejidad temporal de ambas presentaciones también es en función de la cantidad de nodos y ejes del grafo. El análisis es similar a DFS y por lo tanto la complejidad final es O(n+m).

Nuevamente el grafo a recorrer corresponde al de la figura. Iniciamos en el nodo "o". Sus nodos adyacentes corresponden a los nodos 1,2 y 3. Se visita el nodo 1 y se registra para visitar el nodo 4. Como aún hay nodos anteriores descubiertos se visitan. Al acceder al nodo 2 no



encuentra nuevos nodos para visitar desde allí. Se pasa al nodo 3, que permite descubrir el nodo 5. Los nodos 1,2 y 3 están a nivel 1 del nodo de origen. Los descubiertos a partir de estos, 4 y 5, se deben visitar y corresponden a los ubicados en el nivel 2. Primero visita el 4 donde descubre al nodo 6. Luego recorre el nodo 5 y no encuentra nuevos nodos por visitar (que no se hayan visitado o anotado antes). Finalmente visita al nodo 6 que se ubica en el nivel 3. No quedan nodos por visitar, por lo tanto el proceso finaliza. En la imagen se muestra sobre los ejes el orden de visita de los mismos.

Como se observa tanto DFS como BFS finalizan al recorrer todos los nodos del grafo. Estos los establece como métodos de búsqueda exhaustiva. A medida que van recorriendo los nodos generan un **árbol de búsqueda** donde cada uno de ellos es visitado solo una vez desde un nodo antecesor. La estructura final de este árbol es la principal diferencia entre ambos métodos. Por otro lado, su complejidad es similar. Utilizar uno u otro solo corresponde a conveniencia por el tipo de problema a resolver.