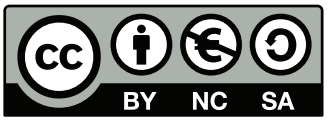
Algorítmica

LibreIM

Doble Grado de Informática y Matemáticas
Universidad de Granada

libreim.github.io/apuntesDGIIM



Algorítmica

LibreIM

Doble Grado de Informática y Matemáticas
Universidad de Granada

libreim.github.io/apuntesDGIIM

Índice

1 Algorítmica

1.1 Introducción.

La ciencia de la computación es el estudio de los algoritmos, incluyendo sus propiedades, hardware, aspectos lingüísticos y aplicaciones.

Podemos definir un **Algoritmo** como una secuencia ordenada y finita de pasos no ambiguos tales que al llevarse a cabo, dará como resultado que se realice la tarea para la que se ha diseñado en un tiempo finito y con recursos limitados.

De esta forma, no podemos por ejemplo obtener la sucesión de **Fibonacci** mediante un algoritmo pues esta es infinita.

Nota : Debemos tener claro que hay una diferencia entre un programa y un algoritmo. Un programa es una serie de instrucciones ordenadas codificadas en un lenguaje de programación que *expresa* un algoritmo.

1.1.1 Los algoritmos.

Un algoritmo es una secuencia finita y ordenada de pasos, exentos de ambigüedad, tal que al llevarse a cabo dará el resultado esperado a la tarea esperada. Se resuelve con recursos limitados y tiempo finito.

- *Definición de Bazaara* : algoritmo como proceso iterativo.
- *Definición de Knuth* :
  Un método computacional es una cuaterna (Q, I, Ω, f) en la que Q es un conjunto que contiene a I y a Ω como subconjuntos y f es una función de Q en Q tal que:

$$f(q) = q \quad \forall q \in \Omega$$

y donde Q es el conjunto de estados del cálculo, I la entrada , Ω la salida y f la regla de cálculo aplicada. Cada entrada $x \in I$ define una sucesión computacionel $x_0, ..., x_n$ tal que $x_0 = x$ y $x_{k+1} = f(x_k)$ si $k \geq 0$.

Se dice que la sucesión computacional termina en k etapas si k es el menor entero para el cual x_k está en Ω y en esa caso, a partir de x se obtiene como salida x_k.

Ejemplo: Algoritmo de Euclídes Dados dos enteros m y n, calcular su m.c.d:

- *E1* : Calculo del resto de m entre n.
- *E2* : Si el resto es 0, el algoritmo termina y la solución es n.
- *E3* : Tomo m=n y n=r, vuelvo a *E1* y repito el proceso.

1.1.2 Propiedades de los algoritmos

Los algoritmos tienen un conjunto de propiedades comunes a todos ellos:

- *Finitud.* Se puede completar en un número finito de pasos.
- *Especifidad.* Cada etapa del algoritmo está definida y descrita correctamente para dar respuesta a una parte determinada del problema.
- *Input.* El algoritmo recibe una entrada de datos.
- *Output.* El algoritmo devuelve un resultado en base a estos datos.
- *Efectividad.* Se completa en un tiempo determinado, que viene dado por el número de operaciones que se requieren para afrontar el problema.

Además los algoritmos dependen tanto de sus propiedades de forma y matemáticas, como del hardware en el que sean ejecutadas. Según el uso que se le quiera dar al algoritmo se implementará en un lenguaje de programación o en otro.

1.1.3 Etapas en la elaboración de un algoritmo

1. *Construcción:* Acto de crear un algoritmo.
2. *Expresión de algoritmos:* Cada paso debe de describirse de manera clara.
3. *Validación:* Que calcule los resultados esperados al problema dado. Es una frase previa a la escritura del programa.
4. *Análisis:* Comprobar el tiempo que tardaría, así como el tamaño que ocuparía. Permite realizar comparaciones teóricas entre algoritmos.
5. *Test:* Corrección de errores que se detecten y comparación de resultados.

La elección final de un algoritmo dependerá, como ya hemos dicho antes, tanto del número de operaciones que se requieran como de los recursos de hardware disponibles. También son otros factores a considerar la adaptación a cualquier tipo de computador, la simplicidad y elegancia del algoritmo y del coste económico de su realización, así como su eficiencia, rapidez y facilidades de programación.

Para probar un algoritmo y validarlo, interesa probar su comportamiento y funcionamiento en los casos extremos, que prevemos que podría causar errores en su funcionamiento.

Comportamiento: Algoritmos Selección y Ordenación Un mismo algoritmo puede dar diversos tiempos en función de los datos del problema que desee resolver.

Caso inicial: U y V arrays de n elementos. U ordenado de forma ascendente y V de forma descendente.

- *Selección:* Indiferentes U y V. Menos de 15% de diferencia en sus ejecuciones.
- *Inserción:* Para U y 5000 elementos tarda $\frac{1}{5}$ segundos. Para V y 5000 elementos tarda 3 minutos y medio.

1.2 Tema 1: La Eficiencia de los Algoritmos

1.2.1 Cálculo de la eficiencia de un algoritmo

Podemos destacar tres métodos para calcular la eficiencia e un algoritmo:

1. *Enfoque empírico o (a posteri):* Programar los diferentes algoritmos candidatos y ejecutarlos sobre diferentes casos con ayuda de un ordenador.
2. *Enfoque teórico o (a priori):* Trata de determinar matemáticamente la cantidad de recursos necesarios para cada algoritmo como una función del tamaño de los casos considerados.
3. *Enfoque Híbrido:* Determina teóricamente la forma de la función que describe la eficiencia del algoritmo y cualquier parámetro numérico que se necesite se determina empíricamente con un ordenador.

El tiempo de ejecución de un programa depende de:

1. *Input del programa.*
2. *Calidad del código generado por computador.*
3. *Naturaleza y velocidad de instrucciones máquina.*
4. *Complejidad del algoritmo.*

No siempre un algoritmo es mejor cuanto menor sea su tiempo de ejecución por ejemplo:

Sean dos algoritmos que consumen uno n^3 días y otro n^2 segundos, el primero es mejor asintóticamente hablando y el segundo mejor que el primero desde un punto de vista práctico, ya que la constante *oculta* lo hace mejor.

1.2.2 Notaciones O y Ω

Para poder comparar los algoritmos empleando los tiempos de ejecución, se emplea una notación asintótica, según la cual un algoritmo de ejecución T(n) se dice que es de orden O(f(n)), si existe una constante positiva c y un número entero n_0 tales que:

$$\forall\ n \geq n_0 \longrightarrow T(n) \leq cf(n)$$

Así queda claro que cuando T(n) es O(f(n)), lo que estamos dando es una cota superior para el tiempo de ejecución, que siempre referiremos al peor caso del problema en cuestión.

De manera análoga, se introduce la notación $\Omega(n)$.

Se dice que un algorítmo A es de orden $\Omega(f(n))$, donde $f(n)$ es una función matemática, cuando existe una implementación del mismo cuyo tiempo de ejecución $T_A(n)$ es mayor o igual que $k \cdot f(n)$, donde K es constante, para "tamaños de casos grandes". Formalmente:

$$A\ es\ \Omega(f(n)) \Leftrightarrow \exists K \in \Re^+, \exists n_0 \in N : T_a(n) \leq K \cdot f(n) \forall n \leq n_0$$

Es de destacar la simetría de ambas notaciones, mientras que una acota superiormente, la otra lo hace inferiormente. Una de las razones por lo que esto es útil, es porque hay veces en las que un algoritmo es rápido, pero no lo es para los *inputs* por lo que debemos de estar dispuestos a saber lo menos que estamos dispuestos en consumir tiempo para resolver cualquier caso de un problema.

1.2.3 Notación Θ

Se dice que un algoritmo A es de orden $\Theta(f(n))$, donde $f(n)$ es una función matemática, cuando existe una implementación del mismo con órdenes de eficiencia $O(f(n))\ y\ \Omega(f(n))$.

1.2.4 Llamadas a funciones recursivas: Ecuaciones en recurrencias

Para calcular el tiempo de ejecución T(n) de una función recursiva hay que plantear el problema con ecuaciones recurrentes.

Método general: Para resolver la ecuación en recurrencia T(n) = c + T(n-1) podemos expandir la ecuación hasta el caso base. $(n-1)\cdot c + T(1) = (n-1)\cdot c + c$

$T(n) = c \cdot n \leq K \cdot n$. Luego el algoritmo sería de orden O(n).

La ecuación característica: Estudiaremos los siguientes casos de ecuaciones:

- Lineales homogéneas de coeficientes constantes
- Lineales no homogéneas de coeficientes constantes

Teniendo en cuenta cambios de variable y de recorrido.

Reescribiendo T(n) como t_n son del tipo:

$$a_0 t_n + a_1 t_{n-1} + ... + a_k t_{n-k} = 0$$

Pasos:

1. Se considera $t_n = x^n$:

$$a_0 x^n + a_1 x^{n-1} + ... + a_k x^{n-k} = 0$$

2. Se saca factor común x^{n-k}:

$$(a_0 x^k + a_1 x^{k-1} + ... + a_k) x^{n-k} = 0$$

Entonces como c no es cero, el otro término vale cero, obteniendo así la ecuación característica:

$$a_0 x^k + a_1 x^{k-1} + ... + a_k = 0$$

Esta ecuación se resuelve como:

$$t_n = \sum_{i=1}^{r} \sum_{j=0}^{M_j - 1} k_{ij} R_i^n n^j$$

Siendo:
R_i las raíces del polinomio
$k_{ij} coeficientes constantes
r el número de raíces distintas del polinomio característico
M_i la multiplicidad de la raíz R_i del polinomio

1.3 Divide y Vencerás

Los algoritmos que aplican *divide y vencerás* obtienen un mejor tiempo de respuesta a problemas grandes mediante la división del problema en subproblemas de tamaño menor, más fáciles de resolver.

Para poder aplicar esta técnica los problemas deben poder dividirse en uno o más casos equicalentes de tamaño menor, que sean independientes entre sí y que puedan resolverse por separado. Este es el caso de algoritmos como MergeSort, Quicksort o selección (ordenación).

Además es necesario que estas subsoluciones independientes se puedan combinar entre sí para poder dar lugar a la solución del caso inicial. Así como también debe de existir una *condición de parada* (caso base resuelto) o un método básico que resuelva el problema para un tamaño pequeño.

El problema del umbral: Si el tamaño del problema inicial es muy grande puede ser tan costoso gestionar millones de llamadas recursivas como resolver el problema original. Pero para poder saber cual es el umbral debemos comparar tiempos de ejecución entre el algoritmo original y el DyV.

1.4 Algoritmos Greedy

Es una forma de resolver algunos problemas mediante algoritmos que construyen la solución paso a paso, siempre actuando con el mismo criterio fijado y sin modificar nada en la resolución anterior.

Estos tienen por ventajas que son eficientes, fáciles de diseñar y de implementar, pero por el contrario puede que la solución alcanzada no sea óptima, o no encontrar ninguna solución aunque el problema la tenga.

Ejemplos de algoritmos greedy los podemos encontrar en el problema del cambio de la máquina expendedora o en diversos algoritmos de caminos en grafos.

Diseño de algoritmos greedy:
Todos los algoritmos greedy se pueden aplicar siempre que existan:

- **Lista de candidatos** para formar la solución.

- **Lista de candidatos usados**.
- **Función solución** para saber cuando un conjunto de candidatos es solución al problema.
- **Criterio de factibilidad** para saber si un candidato puede formar parte de la solución final.
- **Función de selección** del candidato más prometedor para forar parte de la solución
- **Función objetivo** de minimización/maximización

1.5 Exploración en grafos

En ocasiones no existe un algoritmo conocido que pueda resolver un determinado problema, o no de manera eficiente. En esta situación la resolución para por la exploración directa de todas o una gran parte de las posibilidades para llegar a una solución.

Dada la naturaleza de estos problemas se pueden representar como un grafo. Por esto es necesario conocer cómo explorar este.

1.5.1 Recorridos de profundidad y anchura

Los primeros, estudiados en la asignatura Estructura de Datos, se basan en algoritmos recursivos para visitar cada nodo y sus hijos.

Los recorridos en anchura por su parte se basan en un procedimiento parecido, pero se basan en el uso de una cola en la que para cada nodo visitado se elimina este de la cola, se insertan sus adyacentes, se visitan y se inserta w al final de la cola.

Otros ejemplos de estos recorridos son los recorridos **preorden, inorden y postorden.**

1.5.2 Diseño de algoritmos Backtracking

Para representar el posible estado de un problema y los movimientos/acciones para pasar de un estado a otro podemos utilizar árboles y grafos.

De esta idea nace el **Backtracking**. Que consiste en hacer una búsqueda exhaustiva sobre grafos(árboles) dirigidos y acíclicos, mediante un recorrido en profundidad. Se realiza una poda de las ramas poco prometedoras para acelerar su búsqueda.

La solución del problema se expresa dependiendo de la representación del problema, como tuplas $T = (x_1, x_2, x_3, ..., x_t$

Criterio de parada: Dependiendo del problema el objetivo puede ser encontrar todas las soluciones al problema o alguna de ellas.

Pasos para el diseño de Backtracking:

- Buscar una **representación** del tipo $T = (x_1, x_2, x_3, ..., x_t$

- Diseñar las **restricciones implícitas:** Son los valores que cada valor x_i puede tener para construir la solución.
- Identificar las **restricciones explícitas:** Restricciones externas al proceso de encontrar una solución.
- Diseñar la estructura del **árbol/grafo implícito** que define los estados y transiciones entre estados de búsqueda de soluciones.
- Diseñar la **función objetivo:** Criterio de parada para encontrar la solución o soluciones requeridas.
- Adaptar la estructura general del **procedimiento Backtracking** al problema y resolverlo.

1.5.3 Diseños de algoritmos Branch & Bound

Terminología:

- **Nodo vivo:** Nodo del espacio de soluciones del que no se han generado/-visitado aún todos sus hijos.
- **Nodo en curso:** nodo del que se están generando hijos.
- **Nodo muerto:** nodo del que no se van a generar más hijos porque sea hoja, se pode o no produzca una solución mejor que la solución en curso.

Se diferencia de Backtracking en que se generan todos los hijos del nodo en curso antes de que cualquier nodo vivo pase a ser el nodo en curso, mientras que backtracking analizaba inmediatamente todo nodo generado. Eso hace que requiera una estructura auxiliar para almacenar los nodos vivos (Lista de nodos vivos).

Esto hace que backtracking sea más fácil de implementar y tenga un menor requisito de memoria, mientras que branch & bound es más eficiente.

Componentes:

- **Representación** de la solución en una tupla $T = (x_1, x_2, x_3, ..., x_t$
- **Función objetivo** que determina si la solución actual es óptima.
- **Restricciones implícitas:** Valores de cada x_i
- **Restricciones explícitas:** Las que no dependen de la representación del problema.
- **Función de elección** para seleccionar qué nodo es mejor y considerarlo "*en curso*"
- **Cálculo de cotas** para eliminar partes del árbol que no vayan a generar una solución o para elegir el camino más prometedor.
- **Árbol de búsqueda** para organizar el problema.

Al igual que en backtracking, la solución se construye paso a paso. El proceso acaba cuando no quedan nodos en la lista de nodos vivos.

1.6 Programación Dinámica

Algunos problemas pueden subdividirse en problemas de tamaño 1 unidad menor que el problema original, de modo que, para construir la solución a un problema de tamaño N es necesario saber la solución del problema de tamaño N-1.

Ejemplos de este tipo de problemas pueden ser el de los caminos mínimos o el de la mochila.

La programación dinámica se puede usar para encontrar una solución óptima a este tipo de problemas siempre y cuando se cumplan unos requisitos.

Características:

- Se resuelven por etapas, como greedy.
- Divide el problema en subproblemas, como Divide y Vencerás.
- Suele ser una técnica ascendente, es decir, necesitamos conocer la solución del problema de tamaño inmediatamente anterior.
- Mantiene en memoria las soluciones de los subproblemas para ahorrar cálculos repetidos.
- Devuelve la solución óptima ya que para resolver un problema con Programación dinámica se ha de cumplir el **Principio de optimalidad de Bellman**

Diferencias con Divide y vencerás:

- DyV se aplica cuando los problemas son independientes, mientras que PD cuando estos se solapan.
- DyV está basada en recursividad, lo cual ralentiza el proceso, mientras que PD consume más memoria pero evita esta.
- DyV repetirá muchos cálculos mientras que PD mantiene en meoria las subsoluciones para evitar repetir cálculos.

Diferencias con Greedy:

- Greedy en cada etapa elige un elemento y genera una única subsolución, mientras que PD selecciona un elemento pero genera múltiples caminos de etapas a seguir.
- Greedy es eficiente en tiempo y memoria, pero puede no dar una solución o que esta no sea óptima, mientras que PD sólo es eficiente en tiempo, pero a cambio asegura optimalidad.

1.6.1 Diseño de algoritmos de PD

- Son **problemas de optimización** (maximización/minimización).
- El problema debe de poder **resolverse por etapas**
- El problema debe de poder modelarse mediante una **ecuación recurrente**
- Debe existir uno o varios **casos base** al probllema.

- Debe de cumplir el **Principio de Optimalidad de Bellman**.

Este principio se basa en que si una secuencia de pasos para resolver un problema es óptima, entonces cualquier subsecuencia de pasos también es óptima.

2 Algoritmos & Problemas

2.1 Divide y Vencerás

2.1.1 Mergesort

1. Hay un vector de tamaño n.
2. Lo dividimos en dos subvectores de tamaño n/2, estos serán ordenados independientemente.
3. Se combinan los subvectores para generar el vector ordenado del tamaño original n.
4. La condición de parada para dejar de dividir es obtener un vector de tamaño $n \leq 1$

```
1 void mergesort(int* v, int ini, int fin){
2   if(ini < fin){
3     int med = (ini + fin) / 2;
4     mergesort(v,ini,med);
5     mergesort(v,med+1,fin);
6     Combina(v,ini,med,fin);
7   }
8 }
```

Combina() es una función O(n) que se encarga de recorrer los dos subvectores ordenados introduciendo en un nuevo vector el mínimo de ambos para generar el vector de tamaño inicial ordenado.

2.1.2 Quicksort

Es el mejor algoritmo de ordenación en el caso promedio, está basado en la ordenación de un vector de tamaño n del siguiente modo:

1. Determinamos un *pivote*, para dividir el vector en dos partes con elementos menores que el pivote y otra con los elementos mayores o iguales
2. Se ordenan los dos subvectores generados
3. Combinar las dos soluciones para obtener v ordenado

```
1 void QuickSort(int* v, int ini, int fin){
2   int pospivote;
3   if(in < fin){
```

```
4       pospivote = Pivotar((v, ini, fin);
5       QuickSort(v, ini, pospivote - 1);
6       QuickSort(v, pospivote + 1, fin);
7     }
8   }
```

Pivotar es una función que toma como pivote el primer elemento del vector y pasa a la parte izquierda los elementos menores o iguales y a la derecha los elementos mayores que el pivote. Finalmente devuelve la posición central donde se dividen ambas partes.

```
1  // Función para dividir el array y hacer los intercambios
2  int Pivotar(int* v, int start, int end) {
3      int left, right, pivot, temp;
4
5      pivot = array[start];
6      left = start;
7      right = end;
8
9      // Mientras no se cruzen los índices
10     while (left < right) {
11         while (array[right] > pivot) {
12             right--;
13         }
14
15         while ((left < right) && (array[left] <= pivot)) {
16             left++;
17         }
18
19         // Si todavía no se cruzan los indices seguimos
               intercambiando
20         if (left < right) {
21             temp = array[left];
22             array[left] = array[right];
23             array[right] = temp;
24         }
25     }
26
27     // Los índices ya se han cruzado, ponemos el pivot en el
           lugar que le corresponde
28     temp = array[right];
29     array[right] = array[start];
30     array[start] = temp;
31
```

```
32      // La nueva posición del pivot
33      return right;
34  }
```

2.2 Greedy

2.2.1 Problema del Cambio de monedas

Una máquina expendedora tiene que devolver el cambio con un número mínimo de monedas, para esto la máquina tiene el siguiente procedimiento:

1. Mientras quede importe a devolver se coge la mayor moneda disponible y comprueba si es menor que el importe a devolver. Si esta es mayor coge la siguiente moneda más pequeña y se vuelve a empezar. Si es menor, se procede al siguiente paso. Si el importe no es cero y se acaban las monedas o tipo de monedas se devuelve el error “sin solución”.
2. Se añade la moneda al la estructura de datos que almacene la solución. Se resta al importe restante el valor de la moneda y se vuelve al primer paso.
3. En el caso de que el importe restante sea 0 se devuelve la solución calculada.

2.2.2 El problema del arbol generador minimal (AGM)

Sea G=(V,A) un grafo no dirigido conexo, ponderado con pesos no negativos, siendo V el conjunto de vértices del grafo y A el conjunto de aristas. El problema consiste en obtener un grafo parcial, conexo y acíclico tal que la suma de sus aristas sea mínima

Kruskal Empezamos con un conjunto de aristas vacío. En cada paso iremos construyendo la solución final seleccionando la arista de menor coste entre los candidatos. Si dicha arista no forma ciclos, será añadida a la solución, en caso contrario es descartada.

Pseudocódigo:

```
1  Kruskal(grafo G=(V,A)){
2    Ordenar A por orden creciente de pesos
3    N = Número de vértices en V
4    T= vacío (solución a construir)
5    Repetir:
6      a = Arista de A más de menos peso no considerada
7      A = A\{a}
8      Si (T U {a} no forma ciclos)
9        T= T U {a}
10   Hasta que número de aristas en T sea igual a N-1
11   Devolver S
12 }
```

Kruskal devuelve una solución óptima al problema del AGM (demostrable por inducción).

Prim En este caso la solución original será un conjunto de aristas vacío. Pero en cada paso iremos construyendo la solución final seleccionando la arista de menor coste entre las aristas que unen un nodo de la lista de candidatos utilizados con otro de los aún no usados. Así hasta que la lista de candidatos tenga el mismo tamaño que el número de vértices del grafo.

Pseudocódigo:

```
1  Prim(grafo G=(V,A)){
2    B={elemento cualquiera de V} // Candidatos usados
3    T= vacío // Solución a crear
4
5    Mientras(|B| != |V|) hacer:
6      Seleccionar arista a=(b,v) de peso mínimo que una un nodo
           b en
7      B y otro nodo v en V\B
8      T = T U {a}
9      B = B U {v}
10   Fin-Mientras
11
12   Devolver T
13 }
```

2.2.3 El problema de caminos mínimos

Sea G=(V,A) un grafo dirigido, ponderado con pesos no negativos, siendo V el conjunto de vértices del grafo y A el conjunto de aristas. El problema consiste en obtener un conjunto de secuencias de nodos/aristas que definan un camino mínimo entre un nodo origen y todos los demás nodos del grafo.

Algoritmo de Dijkstra: Suponemos que los nodos están numerados entre 0 y n-1. Y la solución serán dos vectores P y D.

P[i] contiene el elemento anterior por el que hay que pasar en el camino mínimo entre el nodo inicial S dado y el nodo i.

D[i] contiene la distancia existente para el camino mínimo entre el nodo inicial S dado y el nodo i.

Existe una matriz L , donde cada componente L[i][j] indica el coste de viajar desde el nodo i al nodo j directamente (matriz de adyacencia).

Metodología a seguir en el algoritmo:

En un primer lugar supondremos que el camino mínimo entre dos nodos S e i es mínimo si L[S][i]>0 (es decir, están conectados directamente). En otro caso suponemos que la distancia es infinita e inicializamos P y D con estos datos.

En cada caso el algoritmo greedy selecciona el nodo v de menor distancia D[v], y comprobamos si para el resto de los nodos w es más corta la distancia D[w] (ir de S a w) o D[v] + L[v][w] (ir de S a w pasando por v). En tal caso se actualiza P y D.

Pseudocódigo:

```
1  Dijkstra(G=(V,A),L,S){
2    C=V\{S} // Candidatos no utilizados
3    Para i = 1 hasta n, hacer:
4      D[i] = L[S][i];
5      P[i] = S;
6    Fin-Para
7
8    Repetir n-1 veces:
9      v=elemento de C tal que D[v] es mínimo // seleccionar el
10                                          // arista de menor
                                               peso
11     C = C\{v}
12
13     // Actualizamos si el camino más corto a los demás nodos
          es el
14     // directo o el que pasa por el nodo que acabamos de
          calcular
15     Para cada w en C, hacer:
16       Si D[w]>D[v]+L[c][w], entonces:
```

```
17          D[w] = D[v]+ L[v][w];
18          P[w] <-- v
19        Fin-Si
20      Fin-Para
21    Fin-Repetir
22    Devolver D,P
23  }
```

2.2.4 El problema del viajante de comercio

Un comerciante quiere distribuir su mercancía entre diferentes ciudades. Su problema consiste en encontrar la ruta más corta tal que:

- Pase por todas las ciudades
- Al finalizar el recorrido, se encuentre en la ciudad de origen
- No debe de pasar por la misma ciudad dos veces

Dicho **formalmente**:

Dado un grafo G=(V,A) no dirigido, completo, con V vértices y A aristas, donde las aristas están ponderadas con pesos no negativos. Encontrar el circuito hamiltoniano minimal del grafo G.

Pseudocódigo:

```
1  TravelingSalesmanProblem(G=(V,A)){
2    C = A // Candidatos no utilizados
3    T = vacío
4    Repetir hasta que |T| = |V|-1:
5      a = (u,v) arista de C con peso mínimo
6      C = C\{a}
7      Si T U {a} no forma ciclos ni v está en 2 aristas de T ni
           y u
8      está en 2 aristas de T entonces:
9        T = T U {a}
10     Fin-Si
11   Fin-Repetir
12
13   A = Arista restante que queda para cerrar el circuito
14   T = T U {a}
15
16   Devolver T
```

Intuitivamente buscamos el circuito minimal (seleccionamos las aristas de menor peso), que sólo tengan "una entrada y una salida" (el arista escogido no puede tener vértices que ya estén en dos aristas) y hamiltoniano (cuando sólo nos quede un arista por escoger, escogemos el que cierre el circuito al nodo donde empezamos).

2.2.5 El problema de la mochila

Tenemos una mochila y un conjunto de objetos a transportar. Y se desea saber qué objetos y en qué cantidad hay que transportarlos para rentabilizar el esfuerzo sabiendo que:

- La mochila tiene un máximo de peso M
- Hay n objetos diferentes a transportar
- Cada objeto i tiene un peso asociado w_i
- Llevar el objeto nos reporta un beneficio b_i
- Incluiremos en la mochila una cantidad x_i del objeto i

De nuevo, **formalmente**:
Maximizar

$$\sum_{i=1}^{n} b_i x_i$$

sujeto a la restricción

$$\sum_{i=1}^{n} w_i x_i \leq M$$

Variantes del problema:

- **Caso continuo:** Todos los objetos son fraccionables
- **Caso discreto:** Ningun objeto es fraccionable
- **Caso del problema 0/1:** Caso particular del discreto, sólo podemos llevar un único ejemplar de cada tipo de objeto.

Problema de la mochila continuo Supondremos que las cantidades que podemos llevar de cada objeto es un número real normalizado al intervalo [0,1] (No llevarnos nada/llevarnos todo el objeto/llevarnos un porcentaje del total del objeto).

Pseudocódigo:

```
1  MochilaContinuo(M, B[0..n-1], W[0..n-1]){
2    C= Conjunto de objetos posibles
3    T= Vector de n cantidades a llevar a 0 // Solución a crear
4    Mientras (suma(peso de llevar las cantidades T)<M) y
5    queden candidatos en C, hacer:
6      Seleccionar i= mejor objeto restante en C
7      C= C\{i}
8
9      Si suma(peso de llevar las cantidades T)+ w_i <= M
10     entonces
11       T_i = 1
12     En otro caso
13       T_i = (M-suma(peso de llevar las cantidades de T))/w_i
14
15   Fin-Mientras
16   Devolver T
17 }
```

2.2.6 El problema de la planificación de tareas

Supongamos que hay n tareas pendientes. Se plantea el problema de la minimización del tiempo que cada tarea espera hasta haberse finalizado.

Sean n tareas pendientes por ejecutar. Se desea encontrar el orden de ejecución de dichas tareas para minimizar el tiempo T que todas las tareas esperan antes de finalizar su ejecución:

$$T = \sum_{i=1}^{n} (Tiempo\ transcurrido\ antes\ de\ que\ i\ finalice)$$

En este problema, cuando se ejecutan las tareas en orden ascendente de tiempo de ejecución obtenemos la mejor planificación. Este diseño sería óptimo.

2.3 Exploración en grafos

2.3.1 Backtracking

Procedimiento general Backtracking:

Pseudocódigo:

```
1  Backtracking(k,T[1, ... , t]){
2    Para cada valor posible de x k hacer:
3      Si es factible T U {x k } entonces
4        Si T U {x k } es solución entonces
5          Devolver T U {x k }
6        En otro caso
7        Si k<t entonces
8          u= BackTracking(k+1, T U {x k })
9          Si u es solución entonces
10           Devolver u
11         Fin-Si
12       Fin-Si
13       Fin-En otro caso
14     Fin-Si
15   Fin-Para
16 Devolver No hay solución
17 }
```

2.3.2 El problema de las 8 reinas

En un tablero de ajedrez de tamaño NxN, se desea colocar N reinas sin que ningún par se dé jaque entre sí.

- Representación: $T = (x_1, x_2, x_3, ..., x_t$ es un vector donde cada componente representa una columna del tablero y cada valor x_i es la fila donde se colocará la reina de la i-ésima columna.
- Restricciones implícitas: x_i tendrá valores entre 1 y N, la dila donde se colocará la i-ésima reina
- Restricciones explícitas: No puede haber 2 reinas en la misma fila, columna o diagonal.
- Función objetivo: Encontrar una tupla $T = (x_1, x_2, x_3, ..., x_t$ que sea solución al problema (N reinas colocadas en el tablero sin darse jaque).
- Función de poda: Al hacer $T = (x_1, x_2, x_3, ..., x_{k-1}$ U x_k , x_k debe cumplir:

Implícitamente por la representación usada para T, no puede hacer 2 reinas en la misma columna.

No existe x_i , con i<k, tal que $x_i = x_k$. (Dos reinas en la misma fila).

No existe x_i, con i<k, tal que abs $(x_i - x_k)$ = abs(i-k).

Pseudocódigo:

```
1  BTNReinas(k,T[1,  ... , N]){
2
3    Para i = 1...N, hacer:
4      T[k] = i
5      Si T es factible entonces
6        Si k = N entonces
7          Devolver T
8        En otro caso:
9          Si k < N entonces
10           T = BTNReinas(k+1,T)
11           Si T es solución entonces
12             Devolver T
13           Fin-Si
14         Fin-Si
15       Fin-En otro caso
16     Fin-Si
17   Fin-Para
18
19   Devolver No hay solución
20 }
```

2.3.3 El problema de la mochila 0/1

Tenemos una mochila con una capacidad de peso máxima M, un conjunto de n objetos a transportar. Cada objeto i tiene un peso w_i y llevarlo supone un beneficio b_i.

Debemos seleccionar el conjunto de objetos que nos reporte el máximo beneficio, siendo estos indivisibles (sólo puede llevarse o no llevarse).

Pseudocódigo:

```
1  BTMochila(k,T,MejorSolucion,M,b,w){
2    Si k>n y Beneficio(T) > Beneficio(MejorSolucion) entonces
3      MejorSolucion = T
4    En otro caso hacer:
5      Para j=0 hasta 1 hacer
6        T[k] = j
7        Si(Peso(T1...k) <= M) entonces
8          MejorSolucion = BTMochila(k+1, T, MejorSolucion, M,b,w
               )
9        En otro caso
10         T[k]=0
11       Fin-Para
```

```
12    Fin-En otro caso
13    Devolver MejorSolucion
```

2.3.4 Branch&Bound

Procedimiento general de Branch&Bound:

Pseudocódigo:

```
1  BranchAndBound(nodoRaiz[n]){
2    C= Cola con prioridad vacía
3    Cota= COTA(nodoRaiz)
4    MejorCoste= +infty
5
6    Mientras C no esté vacía, hacer:
7      x= Quitar Primer Elemento de C
8
9      Si coste(x)<MejorCoste entonces
10       MejorCoste= coste(x)
11       solucion= x
12     Fin-Si
13
14     Si se puede, recalcular cota y eliminar de C los
15     elementos con coste > cota
16
17     Para cada nodo v hijo de x con cota(v)<cota, hacer:
18       Insertar v en C
19     Fin-Para
20
21   Fin-Mientras
22
23   Devolver solucion, MejorCoste
```

2.3.5 El problema de la asignación de tareas

Sean n tareas a repartir entre n personas, suponiendo que el coste de que la persona i se asigne a la tarea j es c_{ij}. Asignar las tareas a las personas de modo que el coste total sea mínimo.

La solución consiste en establecer en primer lugar unas cotas, superior e inferior. La primera puede ser una solución cualquiera a partir de la cual distinguir ramas prometedoras de ramas no prometedoras. La inferior puede ser una situación no factible, pero que sirva de referencia. En cada nodo del primer nivel analizamos la cota mínima suponiendo que le hemos asignado a la primera persona una tarea distinta, entonces descartamos los nodos que no sean prometedores, metiendo en la lista de nodos vivos sólo aquellos cuyas cotas mínimas sean inferiores a la cota superior actual.

2.3.6 El problema de la mochila 0/1

Tenemos una mochila con una capacidad de peso máxima M, un conjunto de n objetos a transportar. Cada objeto i tiene un peso w_i y llevarlo supone un beneficio b_i.

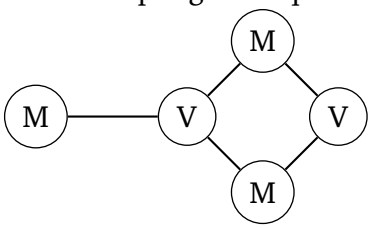
Debemos seleccionar el conjunto de objetos que nos reporte el máximo beneficio, siendo estos indivisibles (sólo puede llevarse o no llevarse).

La diferencia en la resolución de este problema frente al enfoque de Backtracking reside en que Backtracking no hará uso de cotas para descartar ramas.

3 Algoritmos Greedy

Ejercicio 3.1. Sea G un grafo no dirigido y conexo y M un número máximo de colores. Encontrar la asignación de colores mínima entre los colores y los nodos del grafo de tal forma que no existan dos vértices adyacentes con el mismo color.

Solución. Supongamos que tenemos el siguiente grafo:



Veamos los elementos de un algoritmo Greedy aplicados a este problema:

- Lista de candidatos: los M colores.
- Lista de candidatos utilizados: los colores usados.
- Función de selección: escoger un color cualquiera.
- Función de factibilidad: dos ndos adyacentes no pueden tener el mismo color.
- Función solución: todos los nodos están rellenos.
- Función objetivo: minimizar el número de colores a utilizar.

```
1    C ← M
2    K ← ∅
3    T[i] = ∅  ∀i // Sin color en ningún nodo
4
5    Mientras (∃ i T[i] = o y K < C) {
6      K ← K + 1 // Escojo el siguiente color
7      Para cada nodo j ∈ V : T[j] = ∅
8      Si ∄w ∈ V ∃a = (j,w) ∈ A ^ T(w) = k.
```

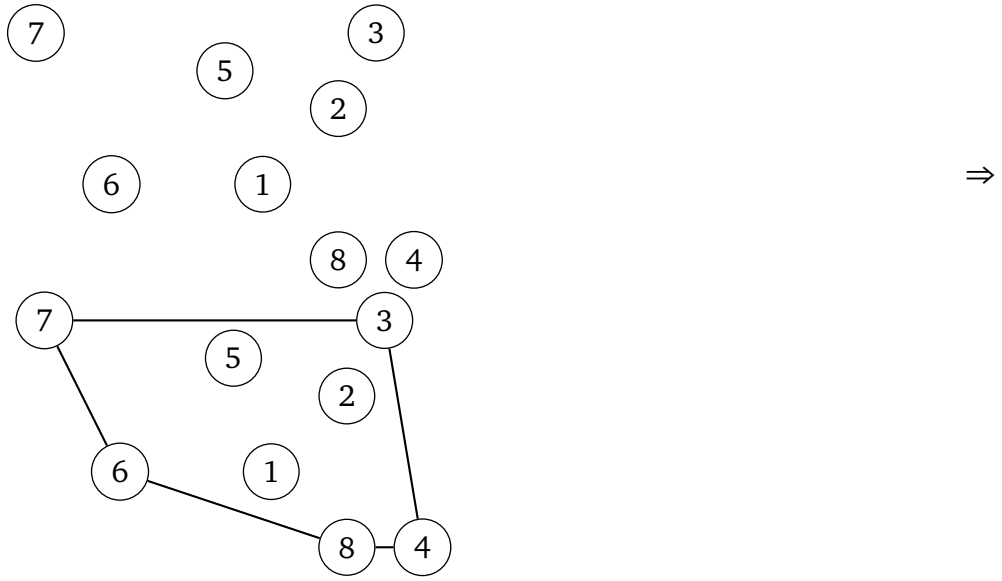
```
 9        T[j] = K
10      }
11      Devolver T
```

Ejercicio 3.2.

Ejercicio 3.3. Dado un conjunto de puntos de planos $p_i = (x_i, y_i)$ que forman un grafo $G = (P, A)$ no dirigido y completo, se desea encontrar el mínimo conjunto de aristas que forman la *envolvente convexa*.

Solución.

Veamos gráficamente lo que representa la *envolvente conexa*. A la izquierda tenemos un conjunto de nodos y a la derecha, la envolvente conexa de este conjunto de nodos.



Diseñemos el algoritmo Greedy para resolver este problema. Enumeremos los elementos de este tipo de algoritmos:

- Lista de candidatos: el conjunto de puntos.
- Lista de candidadatos usados: el conjunto de puntos insertados en la solución.
- Función solución: que no queden candiadatos válidos.
- Criterio de selección: seleccionar un punto P tal que la recta que une P con un punto en la solución deje a todos los puntos a un lado de la recta.
- Criterio de factibilidad: siempre se cumple dado el criterio de selección escogido.
- Función objetivo: minimizar aristas que forman la *envolvente convexa*.

$f(c, t, p)$ es una función que traza la recta entre c y y. Devuelve 0 si p está en

la recta, > 0 si está por encima y < 0 si está por debajo. El pseudocódigo de este algoritmo es el siguiente:

```
1     T = {(p)} ← Greedy (P es el conjunto de puntos p_i = (x_i, y_i).
2
3     V ← Seleccionar nodo de valor y máximo.
4     C ← P \ {v}.
5     T ← ∅
6
7     Mientras queden candidatos válidos {
8       c ← nodo en C : ∃t ∈ T donde ∀p ∈ P f(c,t,p) ≥ 0 ∨ f(c,t,p) ≤ 0
            .
9       C ← C \{c}.
10      T ← T ∪{c}.
11    }
12
13    Ordenar T en el sentido de las agujas del reloj.
14    Devolver T.
```

```
1     {
2       wha
3     }
```