Algorítmica

Tema 3. Algoritmos voraces (*greedy*)

Francisco Javier Cabrerizo Lorite

Curso académico 2023-2024

ETS de Ingenierías Informática y de Telecomunicación. Universidad de Granada

Índice

1. Introducción a los algoritmos voraces
2. Diseño de algoritmos voraces
3. El problema de la mochila
4. Asignación de tareas
5. Planificación de tareas
6. Introducción a la teoría de grafos
7. Árbol de recubrimiento de coste mínimo
8. Caminos de coste mínimo
9. El coloreo de un grafo
10. El viajante de comercio
11. Consideraciones finales

Objetivos

- Ser capaz de proponer diferentes soluciones para un determinado problema y evaluar la calidad de estas.
- Ser consciente de la importancia del análisis de la eficiencia de un algoritmo como paso previo a su implementación.
- Entender la técnica voraz (*greedy*) de resolución de problemas y los distintos casos que se pueden presentar: solución óptima, solución no óptima, o no obtención de la solución.
- Conocer los criterios de aplicación de cada una de las distintas técnicas de diseño de algoritmos.

Introducción a los algoritmos voraces

Introducción a los algoritmos voraces

Greedy

- Sinónimo de voraz, ávido, glotón, etc.

- Son algoritmos no previsores (ya se verá):
  - Pan para hoy y hambre para mañana.
  - Dios proveerá.
  - Reventar antes de que sobre.

Introducción a los algoritmos voraces

Características

- Construyen la solución paso a paso (por etapas).
- En cada etapa «toman el mejor bocado que puedan tragar» sin analizar las posibles consecuencias (voracidad).
- Las decisiones se toman únicamente en base a la información disponible en cada etapa (miopía).
- No vuelven a considerar las decisiones ya tomadas ni las modifican en posteriores etapas.
- Requieren una función objetivo que determine el valor de la solución obtenida.

Introducción a los algoritmos voraces

Uso: problemas de optimización

- Dado un problema con n entradas, se trata de obtener un subconjunto de estas que satisfaga una determinada restricción definida en el problema.
- Cada uno de estos subconjuntos que cumplan las restricciones son soluciones factibles o prometedoras.
- Una solución factible que maximice o minimice la función objetivo se denomina solución óptima.

Introducción a los algoritmos voraces

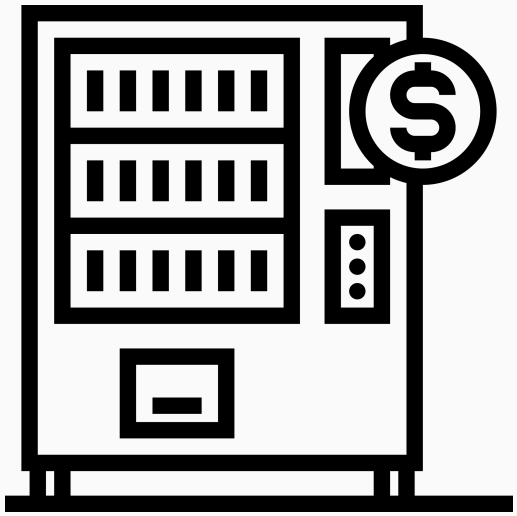
Ventajas

- Eficientes.
- Fáciles de diseñar.
- Fáciles de implementar.

Desventajas

- No alcanzan la solución óptima siempre.
- Pueden no encontrar la solución, aunque esta exista.
- Si no se puede demostrar que alcanzan el óptimo, es un enfoque poco elegante.

«El cambio es inevitable, excepto para una máquina expendedora»



Cambio de monedas

- Supongamos que una máquina expendedora debe devolver un cambio z con el mínimo número de monedas.
- Asumimos que la máquina tiene un número infinito de monedas de céntimos (1, 2, 5, 10, 20, 50) y de euros (1, 2).
- ¿Qué algoritmo seguiría la máquina para resolver el problema?

Formalización

- Sea C las monedas de la máquina, S las monedas a devolver, y s la suma de monedas.
- Se debe devolver el cambio z.

Pseudocódigo

$S \leftarrow \emptyset$
$s \leftarrow 0$
while $s \neq z$ **do**
    $x \leftarrow$ mayor elemento de C tal que $s + x \leq z$
    if $\nexists x$ **then**
        return «No hay cambio»
    else
        $S \leftarrow S \cup \{\text{moneda de valor } x\}$
        $s \leftarrow s + x$
    end if
end while
return S

¿Devuelve el mínimo número de monedas?

Demostrar

- Si no da la solución óptima, basta buscar un contraejemplo.
- Si proporciona la solución óptima, se puede demostrar por reducción al absurdo y por la inducción:
  - Para probar que una proposición es verdadera, se supone que es falsa y se llega a un absurdo o a una contradicción, concluyéndose que la proposición debe ser verdadera, pues no puede ser falsa.
  - Si podemos probar que una proposición es cierta para un primer caso (base de la inducción) y que siempre que sea cierta en un caso también lo debe ser para el siguiente (paso inductivo), entonces es cierta en todos los casos.

¿El algoritmo del cambio de monedas es óptimo?

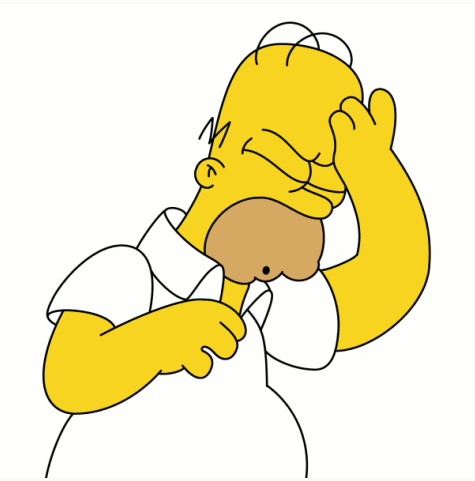
- Depende del sistema monetario.
- $C = \{100, 50, 25, 10, 1\}$ y $z = 40$.

¿El algoritmo del cambio de monedas devuelve siempre solución?

- Depende del número de monedas que queden.
- 3 monedas de 50, 5 monedas de 20 y 12 monedas de 1.
- $z = 63$.

¡Cuidado!

- Dice que no hay solución.
- Sí la hay: 3 monedas de 20 y 3 monedas de 1.



Diseño de algoritmos voraces

Diseño de algoritmos voraces

Elementos de un algoritmo voraz

- **Lista de candidatos**. Representa las posibles decisiones que se pueden tomar en cada momento.
- **Lista de candidatos seleccionados**. Representa las decisiones consideradas y seleccionadas hasta ese momento.
- **Lista de candidatos descartados**. Representa las decisiones consideradas y descartadas hasta ese momento.
- **Función solución**. Determina si los candidatos seleccionados hasta ese momento forman una solución (no necesariamente óptima).
- **Función de factibilidad**. Determina si se puede llegar a una solución (no necesariamente óptima). Se aplica a la lista de candidatos seleccionados junto con el candidato más prometedor.
- **Función selección**. Determina el candidato más prometedor a seleccionar de la lista.
- **Función objetivo**. Determina el valor de la solución alcanzada.

Cambio de monedas

- **Lista de candidatos**: las monedas posibles a devolver.
- **Lista de candidatos seleccionados**: las monedas que se han ido seleccionando para ser devueltas.
- **Función solución**: la suma de las monedas seleccionadas es la cantidad exacta que se debe devolver.
- **Función de factibilidad**: la suma de las monedas actualmente seleccionadas no supera la cantidad que hay que devolver.
- **Función selección**: la moneda de mayor valor que, sumada a las seleccionadas, no supere la cantidad a devolver.
- **Función objetivo**: minimizar el número de monedas que se devuelven para la cantidad z.

Diseño de algoritmos voraces

Funcionamiento de un algoritmo voraz

- Se parte de una lista de candidatos a solución vacía.
- De la lista de candidatos identificados, con la función de selección, se coge el mejor candidato posible (más prometedor).
- Se elimina ese elemento de la lista de candidatos.
- Se comprueba si con ese candidato se puede llegar a construir una solución (si se verifican las condiciones de factibilidad). En tal caso, se añade a la lista de candidatos seleccionados. Si no, se descarta.
- Se evalúa la función solución. Si no hemos terminado, seleccionamos con la función de selección otro candidato y se repite el proceso anterior hasta alcanzar la solución.

Diseño de algoritmos voraces

Esquema

```
function Voraz(candidatos C)
    S ← ∅
    while C ≠ ∅ y no EsSolucion(S) do
        x ← Seleccionar(C)
        C ← C \ {x}
        if EsFactible(S ∪ {x}) then
            S ← S ∪ {x}
        end if
    end while
    if EsSolution(S) then
        return S
    else
        return «No hay solución»
    end if
end function
```

El problema de la mochila

El problema de la mochila

Enunciado

- Considérese una mochila capaz de albergar un peso máximo M y n elementos con pesos $p_1, p_2, \ldots, p_n$ y beneficios $b_1, b_2, \ldots, b_n$.
- Se trata de encontrar las proporciones de los n elementos $x = (x_1, x_2, \ldots, x_n)$, con $0 \leq x_i \leq 1$, que deberán introducirse en la mochila para maximizar el beneficio.
- Se buscan los valores $x_1, x_2, \ldots, x_n$, con $0 \leq x_i \leq 1$, que maximizan la siguiente función objetivo:

$$f(x) = \sum_{i=1}^{n} b_i \cdot x_i$$

con la restricción:

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

El problema de la mochila

Idea general

- Se supone que la mochila está vacía inicialmente.
- En cada paso, se selecciona un elemento a añadir entre los existentes, según algún criterio de selección.
- Insertamos la máxima cantidad posible de ese elemento.
- Se repite el procedimiento hasta que no queden elementos por añadir o hasta que la capacidad de la mochila esté completa.

El problema de la mochila

Criterios de selección

- Escoger la cantidad máxima del elemento de máximo beneficio.
- Escoger la cantidad máxima del elemento de menor peso.
- Escoger la cantidad máxima del elemento con mayor relación beneficio/peso.

El problema de la mochila

- **Lista de candidatos**: los elementos disponibles.
- **Lista de candidatos considerados**: los elementos ya incluidos en la mochila o descartados.
- **Función solución**: no se puede insertar ninguna cantidad de ningún elemento más la mochila (falta de elementos o por peso).
- **Función de selección**: criterio escogido.
- **Función de factibilidad**: el peso de los elementos incluidos no supera la capacidad de la mochila.
- **Función objetivo**: maximizar el beneficio de los elementos a incluir en la mochila sin superar la capacidad de esta.

El problema de la mochila

Pseudocódigo

function Mochila(M, B[1..n], P[1..n])
    $C \leftarrow \{1..n\}$ // Elementos posibles
    $X \leftarrow \{0\}_n$ // Cantidad llevada de cada elemento
    $peso \leftarrow 0$
    while $peso < M$ y $|C| > 0$ **do**
        $i \leftarrow Seleccionar(C)$
        $C \leftarrow C \setminus \{i\}$
        if $peso + P[i] \leq M$ **then**
            $X[i] \leftarrow 1$
            $peso \leftarrow peso + P[i]$
        else
            $X[i] = (M - peso)/P[i]$
            $peso \leftarrow M$
        end if
    end while
    return X
end function

El problema de la mochila

Ejemplo

- Supongamos el siguiente ejemplo ($n = 5$, $M = 100$):

	1	2	3	4	5
p	10	20	30	40	50
b	20	30	66	40	60
b/p	2.0	1.5	2.2	1.0	1.2

- Resultados según el criterio de selección del mejor candidato:

	x_1	x_2	x_3	x_4	x_5	**Beneficio**
Máximo b_i	0	0	1	0.5	1	**146**
Mínimo p_i	1	1	1	1	0	**156**
Máximo b_i/p_i	1	1	1	0	0.8	**164**

El problema de la mochila

Óptimo

- Si se seleccionan los elementos por orden decreciente b_i/p_i, el algoritmo encuentra una solución óptima.

Demostración

- Supongamos, sin pérdida de generalidad, que los elementos disponibles están ordenados de la siguiente manera:

$$b_1/p_1 \geq b_2/p_2 \geq \ldots \geq b_n/p_n$$

- Sea $x = (x_1, x_2, \ldots, x_n)$ la solución dada por el algoritmo voraz:
  - Si $x_i = 1 \ \forall i$, la solución es óptima.
  - En otro caso, sea j el menor índice tal que $x_j < 1$. Por la forma en que trabaja el algoritmo, $x_i = 1 \ \forall i < j$, $x_i = 0 \ \forall i > j$, y además $\sum_{i=1}^{n} p_i \cdot x_i = M$.
- Sea $f(x) = \sum_{i=1}^{n} b_i \cdot x_i$ el beneficio que se obtiene para esa solución.

Demostración

- Sea $y = (y_1, y_2, \ldots, y_n)$ cualquier solución factible, y sea $f(y) = \sum_{i=1}^{n} b_i \cdot y_i$ su beneficio.
- Por ser solución factible, cumple que $\sum_{i=1}^{n} p_i \cdot y_i \leq M$. Entonces, puesto que $\sum_{i=1}^{n} p_i \cdot x_i = M$, restando ambas capacidades, podemos afirmar que:

$$\sum_{i=1}^{n} (p_i \cdot x_i - p_i \cdot y_i) = \sum_{i=1}^{n} p_i \cdot (x_i - y_i) \geq 0$$

- La diferencia de beneficios es:

$$f(x) - f(y) = \sum_{i=1}^{n} b_i \cdot (x_i - y_i) = \sum_{i=1}^{n} \frac{b_i}{p_i} \cdot p_i \cdot (x_i - y_i)$$

Demostración

- Considérese ahora de nuevo el menor índice j tal que $x_j < 1$ en la solución x obtenida mediante el algoritmo voraz propuesto:
  - Si $i < j$, entonces $x_i = 1$, y por tanto $(x_i - y_i) \geq 0$. Además, por la forma en que se han ordenado los elementos, $\frac{b_i}{p_i} \geq \frac{b_j}{p_j}$.
  - Si $i > j$, entonces $x_i = 0$, y por tanto $(x_i - y_i) \leq 0$. Además, por la forma en que se han ordenado los elementos, $\frac{b_i}{p_i} \leq \frac{b_j}{p_j}$.
  - Por último, si $i = j$, entonces $\frac{b_i}{p_i} = \frac{b_j}{p_j}$.
- Así pues, en todos los casos se tiene que: $\frac{b_i}{p_i} \cdot (x_i - y_i) \geq \frac{b_j}{p_j} \cdot (x_i - y_i)$.
- Por tanto:

$$f(x) - f(y) = \sum_{i=1}^{n} \frac{b_i}{p_i} \cdot p_i \cdot (x_i - y_i) \geq \frac{b_j}{p_j} \sum_{i=1}^{n} p_i \cdot (x_i - y_i) \geq 0$$

es decir, $f(x) \geq f(y)$, tal como se quería demostrar.

El problema de la mochila

El problema de la mochila discreto

- El conjunto solución no puede contener fracciones de elementos, es decir, x_i solo podrá ser 0 o 1, $\forall i \in [1, n]$.
- Como en el problema continuo, el objetivo es maximizar el beneficio, definido como $\sum_{i=1}^{n} b_i \cdot x_i$, con la restricción $\sum_{i=1}^{n} p_i \cdot x_i \leq M$.

¿Seguirá en este caso dando la solución óptima el algoritmo voraz propuesto para el caso continuo?

El problema de la mochila

Respuesta

- No, como lo demuestra el siguiente contraejemplo:
  - Supongamos una mochila de capacidad $M = 6$, con los siguientes elementos:

	1	2	3
p	5	3	3
b	11	6	6
b/p	2.2	2	2

  - El algoritmo voraz elige como mejor candidato el primer elemento, y como ya no caben más, la solución tiene un beneficio de 11.
  - Hay una solución mejor, ya que se pueden introducir en la mochila los dos últimos elementos, con un beneficio total de 12.

Asignación de tareas

Asignación de tareas

Enunciado

- Supongamos que disponemos de n trabajadores y n tareas.
- Sea $c_{ij} > 0$ el coste de asignar la tarea j al trabajador i.
- Una asignación de tareas puede expresarse como una asignación de los valores 0 o 1 a las variables x_{ij}:
  - $x_{ij} = 0$ significa que al trabajador i no le han asignado la tarea j.
  - $x_{ij} = 1$ indica que sí.
- Una asignación es válida si a cada trabajador solo le corresponde una tarea y cada tarea está asignada a un único trabajador.
- Dada una asignación válida, se define su coste como:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} \cdot c_{ij}$$

- Diremos que una asignación es óptima si es de mínimo coste.

Asignación de tareas

Criterios de selección

- Asignar a cada tarea el mejor trabajador disponible.
- Asignar a cada trabajador la mejor tarea posible.

¿Es alguna estrategia mejor que la otra?

¿Es alguna óptima?

Asignación de tareas

Criterio de selección

- Asignar a cada tarea T_j el mejor trabajador P_i.

	T_1	T_2	T_3	T_4
P_1	11	12	18	40
P_2	14	15	13	22
P_3	11	17	19	23
P_4	17	14	20	28

- Coste: $11 + 12 + 13 + 28 = \mathbf{64}$

Asignación de tareas

Criterio de selección

- Asignar a cada trabajador P_i la mejor tarea T_j.

	T_1	T_2	T_3	T_4
P_1	11	12	18	40
P_2	14	15	13	22
P_3	11	17	19	23
P_4	17	14	20	28

- Coste: $11 + 13 + 17 + 28 = \mathbf{69}$

Asignación de tareas

¿Alguno de los dos criterios de selección proporciona el óptimo?

	T_1	T_2	T_3	T_4
P_1	11	12	18	40
P_2	14	15	13	22
P_3	11	17	19	23
P_4	17	14	20	28

Respuesta: no

- Coste: $11 + 14 + 13 + 23 = \mathbf{61}$
- Algoritmo húngaro.

Asignación de tareas

Diseño voraz

- **Lista de candidatos**: los trabajadores o las tareas.
- **Lista de candidatos seleccionados**: los trabajadores o las tareas que ya han sido asignados.
- **Función solución**: todos los trabajadores/tareas han sido asignados.
- **Función de factibilidad**: un trabajador/tarea solo puede asignarse una única vez.
- **Función selección**: el trabajador de menor coste para cada tarea o la tarea de menor coste para cada trabajador.
- **Función objetivo**: minimizar el coste total de asignar cada trabajador a cada tarea:

$$\min_{x_{ij}} \left\{ \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} \cdot c_{ij} \right\}$$

Asignación de tareas

Pseudocódigo

function AsignaciónTareas($C[1..n][1..n]$)
    $U \leftarrow \{1, \ldots, n\}$ // Candidatos a asignar
    $X \leftarrow \{0\}_{n \cdot n}$
    for all $i \in [1, \ldots, n]$ **do**
        if Criterio 1 **then**
            $j \leftarrow$ trabajador j donde $C[j][i]$ sea mínimo y $j \in U$
            $X[j][i] = 1$
        else
            $j \leftarrow$ tarea j donde $C[i][j]$ sea mínimo y $j \in U$
            $X[i][j] = 1$
        end if
        $U \leftarrow U \setminus \{j\}$
    end for
    return X
end function

Planificación de tareas

Planificación de tareas

Enunciado

- Supongamos que un único sistema tiene que ejecutar n tareas.
- Sea t_i el tiempo que tarda en ejecutarse la tarea i, $1 \leq i \leq n$.
- Se desea minimizar el tiempo total T invertido en el sistema por todas las tareas (minimizar el tiempo medio invertido por cada tarea en el sistema). Es decir, asumiendo que E_i es el tiempo en el sistema para la tarea i-ésima, se desea minimizar:

$$T = \sum_{i=1}^{n} E_i$$

- Diremos que una planificación es óptima si es de mínimo tiempo total invertido.

Planificación de tareas

Ejemplo

- Supongamos que hay tres tareas pendientes $(1, 2, 3)$ cuyos tiempos de ejecución son $t_1 = 5$, $t_2 = 10$ y $t_3 = 3$.
- Consideremos los siguientes órdenes de ejecución y el tiempo total invertido T asociado a cada uno de ellos:

$$
\begin{aligned}
1, 2, 3 &: & 5 + (5 + 10) + (5 + 10 + 3) &= 38 \\
1, 3, 2 &: & 5 + (5 + 3) + (5 + 3 + 10) &= 31 \\
2, 1, 3 &: & 10 + (10 + 5) + (10 + 5 + 3) &= 43 \\
2, 3, 1 &: & 10 + (10 + 3) + (10 + 3 + 5) &= 41 \\
3, 1, 2 &: & 3 + (3 + 5) + (3 + 5 + 10) &= 29 \\
3, 2, 1 &: & 3 + (3 + 10) + (3 + 10 + 5) &= 34
\end{aligned}
$$

Idea general

- La planificación óptima se obtiene cuando las tareas se ejecutan en orden ascendente de tiempo de ejecución.

Planificación de tareas

Diseño voraz

- **Lista de candidatos**: las n tareas a ejecutar.
- **Lista de candidatos seleccionados**: las tareas ya planificadas.
- **Función solución**: el número de tareas planificadas es n.
- **Función selección**: se selecciona la tarea i tal que su tiempo de ejecución t_i es mínimo.
- **Función objetivo**: minimizar:

$$T = \sum_{i=1}^{n} E_i$$

donde E_i es el tiempo en el sistema para la tarea i-ésima.

Planificación de tareas

Pseudocódigo

function PlanificacionTareas($T[1..n]$)
    $C \leftarrow \{1, \ldots, n\}$
    $P \leftarrow \{0\}_n$
    for all $i \in [1, \ldots, n]$ **do**
        $j \leftarrow$ seleccionar $j \in C$ tal que $T[j]$ sea mínimo.
        $C \leftarrow C \setminus \{j\}$
        $P[i] \leftarrow j$
    end for
    return P
end function

Óptimo

- Si se seleccionan las tareas por orden ascendente de tiempo de ejecución t_i, el algoritmo proporciona la solución óptima.

Demostración

- Sea $P = (p_1, p_2, \ldots, p_n)$ cualquier permutación de enteros del 1 al n, y sea $s_i = t_{p_i}$. Si las tareas se ejecutan en el orden P, entonces el tiempo de ejecución requerido por la i-ésima tarea será s_i, y el tiempo total invertido en el sistema por todas las tareas para la permutación P, $T(P)$, es:

$$\begin{aligned} T(P) &= s_1 + (s_1 + s_2) + \ldots + (s_1 + s_2 + \ldots + s_n) \\ &= n \cdot s_1 + (n-1) \cdot s_2 + \ldots + 2 \cdot s_{n-1} + s_n \\ &= \sum_{k=1}^{n} (n - k + 1) \cdot s_k \end{aligned}$$

Demostración

- Supongamos ahora que P no organiza las tareas por orden de tiempos crecientes de ejecución. Entonces, se pueden encontrar dos enteros a y b con $a < b$ y $s_a > s_b$. Si intercambiamos la posición de estas dos tareas, se obtiene un nuevo orden de ejecución P' que es simplemente el orden P después de intercambiar los enteros p_a y p_b.
- El tiempo total para la planificación P', $T(P')$, es:

$$T(P') = (n - a + 1) \cdot s_b + (n - b + 1) \cdot s_a + \sum_{\substack{k=1 \\ k \neq a,b}}^{n} (n - k + 1) \cdot s_k$$

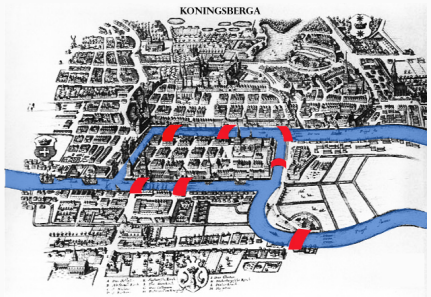
- Si comparamos $T(P)$ con $T(P')$, vemos que la planificación nueva es preferible a la vieja:

$$\begin{aligned} T(P) - T(P') &= (n - a + 1) \cdot (s_a - s_b) + (n - b + 1) \cdot (s_b - s_a) \\ &= (b - a) \cdot (s_a - s_b) > 0 \end{aligned}$$

Introducción a la teoría de grafos

Introducción la teoría de grafos

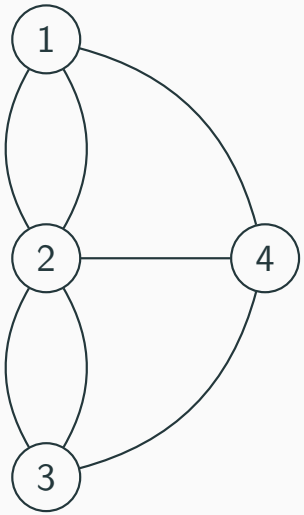
- Surge de un trabajo del matemático suizo Leonhard Euler en 1736, basado en el problema de los puentes de Königsberg.
- Königsberg (Kaliningrado) era famosa por los siete puentes que unían ambos márgenes del río Pregel con dos de sus islas.
- El problema que se planteaba era si era posible, partiendo de un lugar arbitrario, regresar al lugar de partida cruzando cada puente una sola vez.



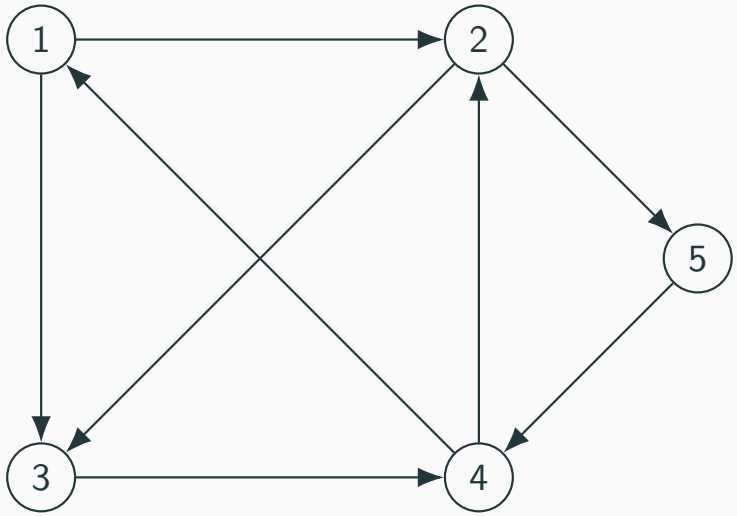
- Demostró que el grafo asociado al esquema de puentes de Königsberg no tiene solución.
- Resolvió un problema más general: las condiciones debe satisfacer un grafo para garantizar que se puede regresar al vértice de partida sin pasar por la misma arista más de una vez.

Introducción la teoría de grafos

- Un **grafo simple** G es un par (V, A) tal que:
  - V es un conjunto finito no vacío de vértices o nodos. Se llama orden de G al número de vértices de G y se denota n.
  - A es un subconjunto del producto cartesiano $V \times V$ (pares de vértices), cuyos elementos se denominan arcos (grafos dirigidos) o aristas (grafos no dirigidos), que conectan los vértices de V.
- En un grafo simple no existen auto ciclos ni multi-aristas.

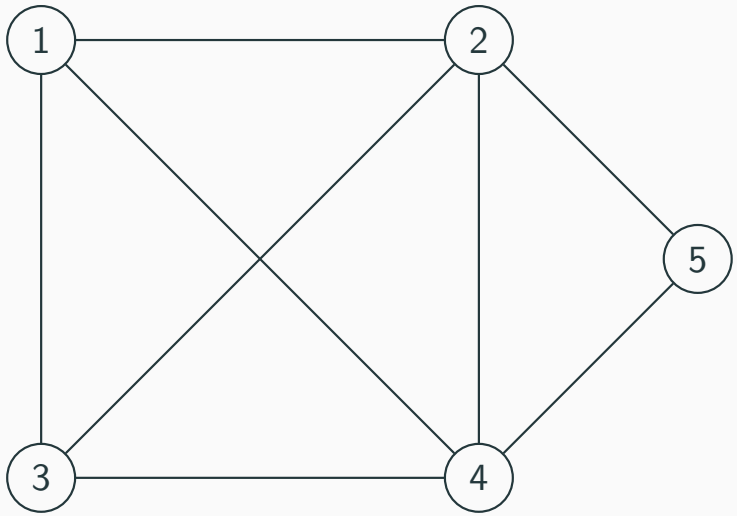


- En un **grafo dirigido**, cada arco viene dado por un par ordenado $\langle u, v \rangle$, donde $u \in V$ es el vértice inicial del arco y $v \in V$ es el vértice final.



- El máximo número de arcos de un grafo dirigido es $n \cdot (n-1)$. En tal caso, se dice que el grafo es completo.

- En un **grafo no dirigido**, no se distinguen los vértices inicial y final. Dados dos vértices, $u, v \in V$, la arista (u, v) conecta dichos vértices en ambos sentidos.

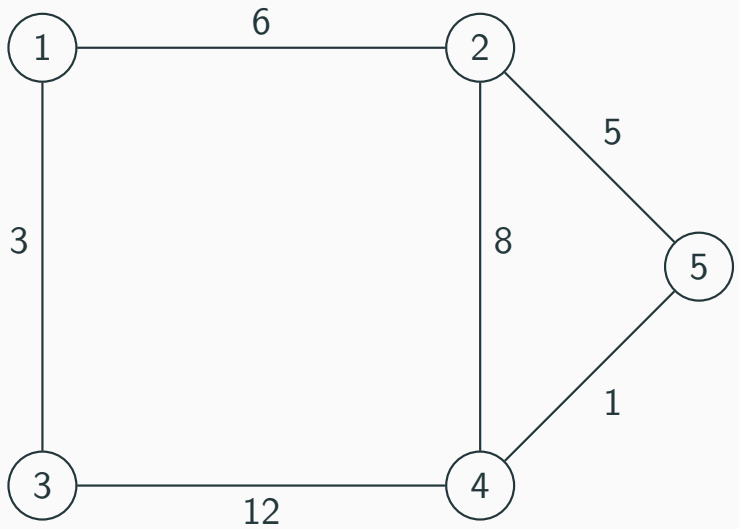


- El máximo número de aristas de un grafo no dirigido es $n \cdot (n-1)/2$. En tal caso, se dice que el grafo es completo.

Introducción la teoría de grafos

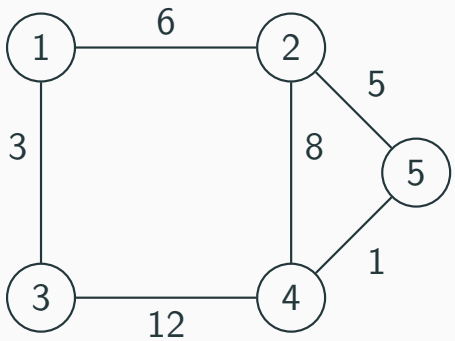
- El **grado de un vértice** es el número de arcos o aristas que inciden en dicho vértice.
- En grafos dirigidos, se definen el **grado de entrada** (número de arcos que entran al vértice) y el **grado de salida** (número de arcos que salen del vértice).
- Dos **arcos** o **aristas** se dice que son **adyacentes** si tienen un vértice en común.
- Dos **vértices** son **adyacentes** si existe algún arco o arista que los una.

- Los **grafos ponderados** son aquellos en los que los arcos o aristas llevan asignado un cierto valor real, denominado **peso**, que representa la distancia o el coste de desplazamiento entre dos vértices adyacentes.

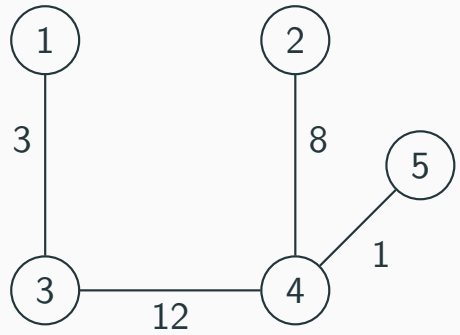


- Llamamos **subgrafo** de $G = (V, A)$ a un grafo $G_b = (V_b, A_b)$ tal que $V_b \subseteq V$ y $A_b = A \cap (V_b \times V_b)$. En el subgrafo G_b están todos los arcos o aristas que hay en A_b y que unen los vértices de V_b.

- Un **camino** entre los vértices u y v de un grafo no dirigido $G = (V, A)$ es una secuencia de vértices $u, w_1, \ldots, w_k, v$ tal que las aristas $(u, w_1), \ldots, (w_k, v) \in A$. En un grafo dirigido, un camino se define de la misma forma, pero usando arcos en lugar de aristas.
- Se denomina **camino simple** a un camino en el cual todos sus vértices son distintos salvo, quizás, el primero y el último.
- Se denomina **ciclo** a un camino simple en el cual los vértices primero y último son iguales.

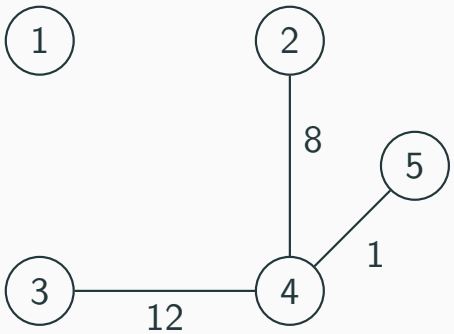


Grafo con ciclos

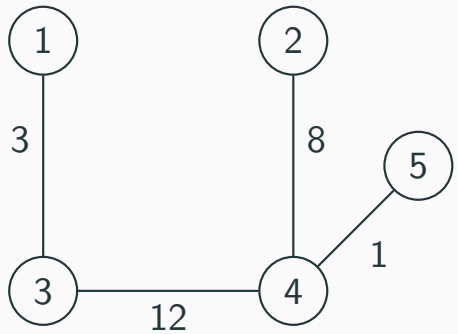


Grafo sin ciclos

- La **longitud de un camino** es el número de arcos o aristas que lo componen.
- En un grafo no dirigido, dos vértices u y v están **conectados** si existe al menos un camino entre ambos. Un grafo no dirigido es **conexo** si existe al menos un camino entre todo par de vértices.
- En un grafo dirigido, un vértice u es **accesible** desde otro vértice v si existe al menos un camino que parte de v y termina en u. Un grafo dirigido es **fuertemente conexo** si todo vértice u es accesible desde cualquier otro vértice v.



Grafo no conexo



Grafo conexo

Representación de grafos

- **Matriz de adyacencia**. Se usa una matriz M de tamaño $n \times n$ (siendo n el número de vértices del grafo) donde las filas y columnas hacen referencia a los vértices y cada elemento $M[i][j]$ almacena o bien un valor booleano que indica la conexión o no entre los vértices i y j, o bien el peso/coste de tal conexión (∞ si no existiera).
- **Lista de adyacencia**. Se usa una estructura L de tamaño n donde $L[i]$ contiene la lista de vértices adyacentes a i y, en el caso de grafos ponderados, también el peso/coste asociado a cada conexión.

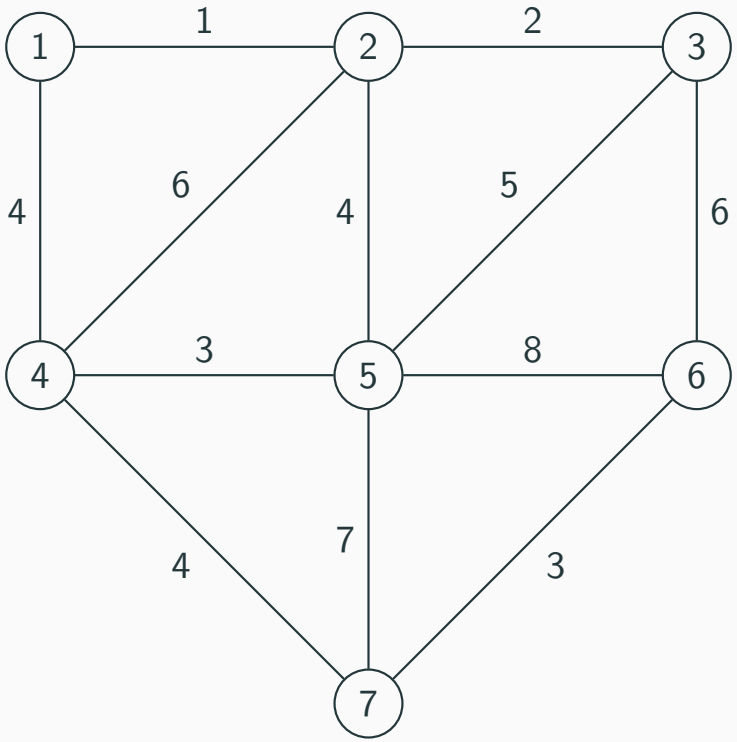
Árbol de recubrimiento de coste mínimo

Árbol de recubrimiento de coste mínimo

Enunciado

- Considérese un grafo no dirigido, ponderado y conexo $G = (V, A, c)$, donde $c : A \to \mathbb{R}^+$ es la función que asigna a cada arista (u, v) un coste $c(u, v) > 0$.
- Se quiere encontrar el árbol de recubrimiento de G de coste mínimo, es decir, el subgrafo conexo acíclico (árbol) que contiene todos los vértices de G tal que la suma de sus aristas sea mínima.

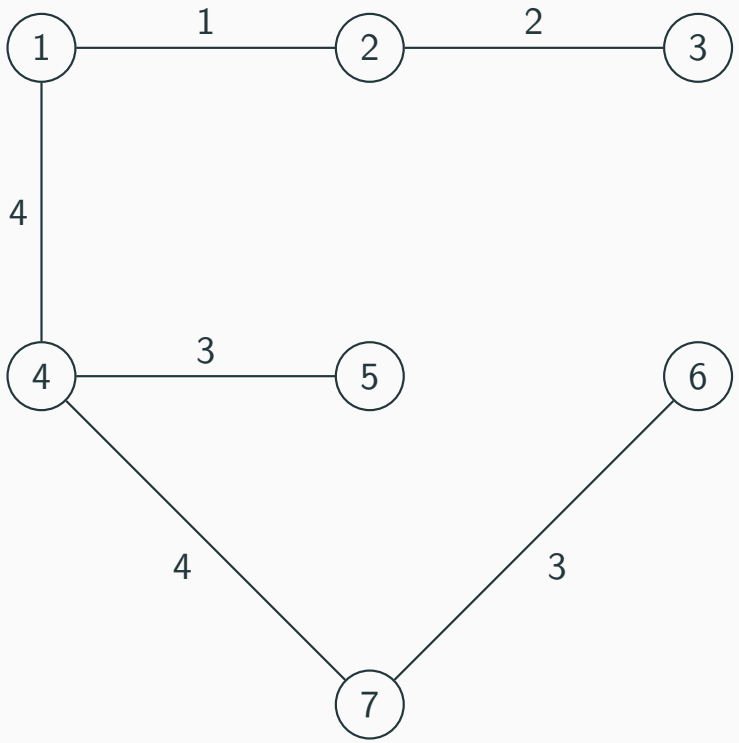
Árbol de recubrimiento de coste mínimo



Inicio

- Grafo no dirigido y conexo.
- Ponderado con pesos mayores que 0.

Árbol de recubrimiento de coste mínimo



Objetivo

- Un grafo parcial acíclico y conexo.
- La suma de los pesos de las aristas debe ser mínima.

Árbol de recubrimiento de coste mínimo

Estrategias de resolución

- Algoritmo de Kruskal (basado en aristas).
- Algoritmo de Prim (basado en vértices).

Algoritmo de Kruskal: idea general

- Inicialmente, la solución será un conjunto de aristas vacío.
- En cada paso, se selecciona la arista de menor coste de entre las aristas que forman la lista de candidatos.
- Si la arista seleccionada no forma ciclos con las ya escogidas, se añade a la solución. En otro caso, se descarta.
- Como en cada paso se selecciona la arista de menor coste, conviene tenerlas ordenadas previamente.

Árbol de recubrimiento de coste mínimo

Algoritmo de Kruskal: diseño voraz

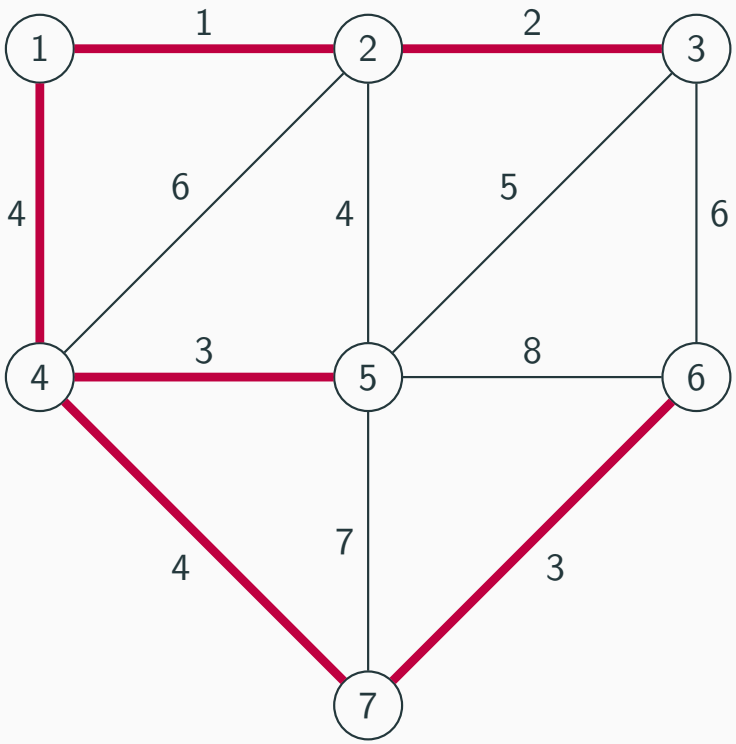
- **Lista de candidatos**: las aristas del grafo.
- **Lista de candidatos seleccionados**: las aristas que se han seleccionado del grafo original.
- **Función solución**: las aristas seleccionadas unen todos los vértices del grafo sin formar ciclos (árbol con $n-1$ aristas).
- **Función de selección**: se selecciona la arista de menor coste.
- **Función de factibilidad**: el conjunto de aristas seleccionadas no forma ciclos.
- **Función objetivo**: minimizar la suma de los pesos de las aristas que forman la solución.

Árbol de recubrimiento de coste mínimo

Algoritmo de Kruskal

function Kruskal($G = (V, A)$)
    ordenar A por orden creciente de coste
    $n \leftarrow$ número de vértices de V
    $T \leftarrow \emptyset$
    Iniciar n conjuntos, cada uno con un elemento distinto de V
    while $|T| < n - 1$ **do**
        $a \leftarrow (u, v) \leftarrow$ arista de menor coste de A
        $A \leftarrow A \setminus \{a\}$
        $compu \leftarrow$ Buscar(u)
        $compv \leftarrow$ Buscar(v)
        if $compu \neq compv$ **then**
            Fusionar($compu, compv$)
            $T \leftarrow T \cup \{a\}$
        end if
    end while
    return T
end function

Árbol de recubrimiento de coste mínimo



Paso	Arista	¿Aceptada?	Componentes conexas
Inicialización	–	–	{1} {2} {3} {4} {5} {6} {7}
1	(1,2)	Sí	{1,2} {3} {4} {5} {6} {7}
2	(2,3)	Sí	{1,2,3} {4} {5} {6} {7}
3	(4,5)	Sí	{1,2,3} {4,5} {6} {7}
4	(6,7)	Sí	{1,2,3} {4,5} {6,7}
5	(1,4)	Sí	{1,2,3,4,5} {6,7}
6	(2,5)	No	{1,2,3,4,5} {6,7}
7	(4,7)	Sí	{1,2,3,4,5,6,7}

Coste de la solución: 17

Árbol de recubrimiento de coste mínimo

Consideraciones

- Un conjunto de aristas factible es **prometedor** si se puede extender para producir no solo una solución, sino una solución óptima al problema del árbol de recubrimiento de coste mínimo.
- El conjunto vacío siempre es prometedor (siempre existe una solución óptima).
- Si un conjunto de aristas prometedor ya es una solución, la extensión requerida es irrelevante, y esta solución debe ser óptima.
- Una arista **sale** de un conjunto de vértices dado si tiene exactamente un extremo en el conjunto de vértices.

Árbol de recubrimiento de coste mínimo

Lema

- Sea $G = (V, A)$ un grafo no dirigido, ponderado y conexo. Sea $B \subset V$ un subconjunto estricto de los vértices de G. Sea $T \subseteq A$ un conjunto de aristas prometedor, tal que no haya ninguna arista de T que salga de B. Sea a la arista de menor coste que sale de B (o una de las de menor coste si hay empates). Entonces, $T \cup \{a\}$ es prometedor.

Demostración del lema

- Como T es prometedor, existe un árbol de recubrimiento de coste mínimo, U, de G, tal que $T \subseteq U$.
- Si $a \in U$, entonces no hay nada que probar.
- Si $a \notin U$, al añadir la arista a al árbol U se creará un ciclo. En este, puesto que a sale de B, existe necesariamente al menos otra arista d que también sale de B, o bien el ciclo no se cerraría.
- Si se elimina d, el ciclo desaparece y se obtiene un nuevo árbol W que recubre G.
- Sin embargo, el coste de a, por definición, no es mayor que el de d y, por tanto, el coste total de las aristas de W no sobrepasa el de las aristas de U. Por tanto, W es también un árbol de recubrimiento de coste mínimo de G y contiene a la arista a.
- La demostración se completa viendo que $T \subseteq W$, ya que la arista d que se ha eliminado sale de B y, por tanto, no podría haber sido una arista de T.

Árbol de recubrimiento de coste mínimo

Óptimo

- El algoritmo de Kruskal obtiene un árbol de recubrimiento de coste mínimo.

Demostración

- Por inducción sobre el número de aristas en T. Si T es prometedor, sigue siendo prometedor en cualquier fase del algoritmo cuando se le añade una arista adicional. Cuando el algoritmo termina, T da una solución al problema. Puesto que también es prometedora, la solución es óptima.
- **Base**. El conjunto vacío es prometedor porque G es conexo y, por tanto, tiene que existir una solución.

Demostración

- **Paso de inducción**. Supongamos que T es prometedor inmediatamente antes de que el algoritmo añada una nueva arista $a = (u, v)$. Las aristas de T dividen a los vértices de G en dos o más componentes conexas. El vértice u se encuentra en una de estas componentes y v está en otra. Sea B el conjunto de vértices de la componente que contiene a u:
  - El conjunto B es un subconjunto estricto de los vértices de G (por ejemplo, no incluye a v).
  - T es un conjunto prometedor tal que ninguna arista de T sale de B.
  - a es una de las aristas de coste mínimo que salen de B (todas las aristas de menor coste ya se han examinado y se han añadido a T o se han descartado porque tenían los dos extremos en la misma componente conexa).
- Entonces, se cumplen las condiciones del lema y se concluye que el conjunto $T \cup \{a\}$ también es prometedor.

Algoritmo de Prim: idea general

- La solución, al igual que en el algoritmo de Kruskal, es un conjunto de aristas.
- Sin embargo, la idea general del algoritmo de Prim es seleccionar vértices, no aristas.
- Inicialmente, la solución será un conjunto de aristas vacío y habrá un vértice inicial como candidato usado.
- En cada paso, se selecciona un vértice para unirlo con los que ya se han seleccionado, bajo el criterio de que la arista que une ese vértice con alguno de los ya seleccionados tenga un coste mínimo.

Árbol de recubrimiento de coste mínimo

Algoritmo de Prim: diseño voraz

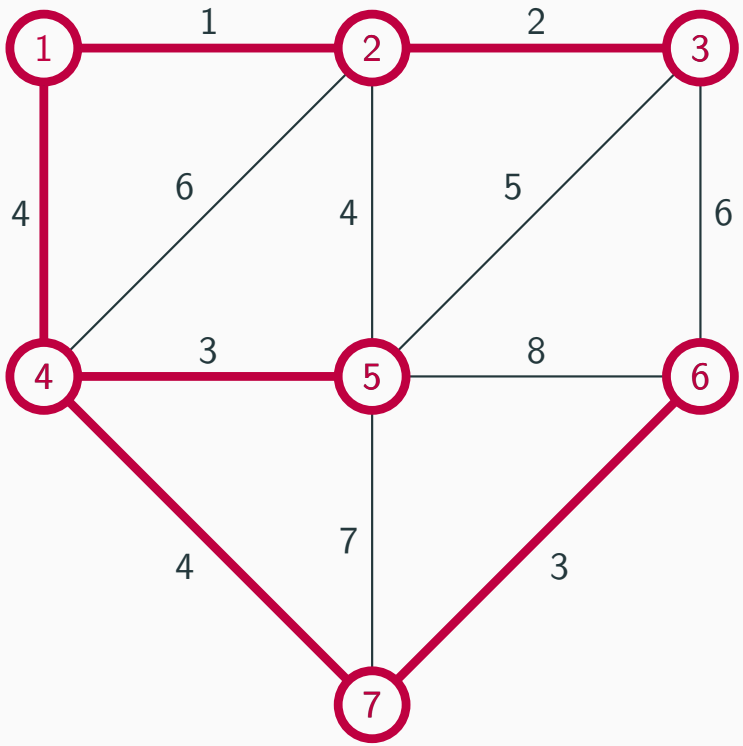
- **Lista de candidatos**: los vértices del grafo.
- **Lista de candidatos seleccionados**: los vértices que se han seleccionado para obtener las aristas que forman la solución.
- **Función solución**: n vértices seleccionados. El conjunto de aristas conecta todos los vértices.
- **Función de selección**: se selecciona el vértice no utilizado tal que la arista que une el vértice con uno ya utilizado tiene coste mínimo.
- **Función de factibilidad**: no deben existir ciclos en las aristas que unen los vértices seleccionados.
- **Función objetivo**: minimizar la suma de los pesos de las aristas que forman la solución.

Árbol de recubrimiento de coste mínimo

Algoritmo de Prim

function Prim($G = (V, A)$)
    $B \leftarrow$ elemento cualquiera de V
    $T \leftarrow \emptyset$
    while $|B| \neq |V|$ **do**
        buscar $a = (u, v)$ de coste mínimo tal que $u \in B$ y $v \in V \setminus B$
        $T \leftarrow T \cup \{a\}$
        $B \leftarrow B \cup \{v\}$
    end while
end function

Árbol de recubrimiento de coste mínimo



Paso	Arista	B
Inicialización	–	{1}
1	(1,2)	{1,2}
2	(2,3)	{1,2,3}
3	(1,4)	{1,2,3,4}
4	(4,5)	{1,2,3,4,5}
5	(4,7)	{1,2,3,4,5,7}
6	(7,6)	{1,2,3,4,5,6,7}

Coste de la solución: 17

Árbol de recubrimiento de coste mínimo

Óptimo

- El algoritmo de Prim obtiene un árbol de recubrimiento de coste mínimo.

Demostración

- Por inducción sobre el número de aristas en T. Si T es prometedor, sigue siendo prometedor en cualquier fase del algoritmo cuando se le añade una arista adicional. Cuando el algoritmo termina, T da una solución al problema. Puesto que también es prometedora, la solución es óptima.
- **Base**. El conjunto vacío es prometedor.

Demostración

- **Paso de inducción**. Supongamos que T es prometedor inmediatamente antes de que el algoritmo añada una nueva arista $a = (u, v)$. Sea $\overline{B} = V \setminus B$ el conjunto de vértices de G que no están en B:
  - El conjunto $\overline{B}$ es un subconjunto estricto de V (ya que B lo es).
  - T es un conjunto prometedor tal que ninguna arista de T sale de $\overline{B}$.
  - a es por definición una de las aristas de coste mínimo que salen de $\overline{B}$ y conectan con B.
- Entonces, se cumplen las condiciones del lema y se concluye que el conjunto $T \cup \{a\}$ también es prometedor.

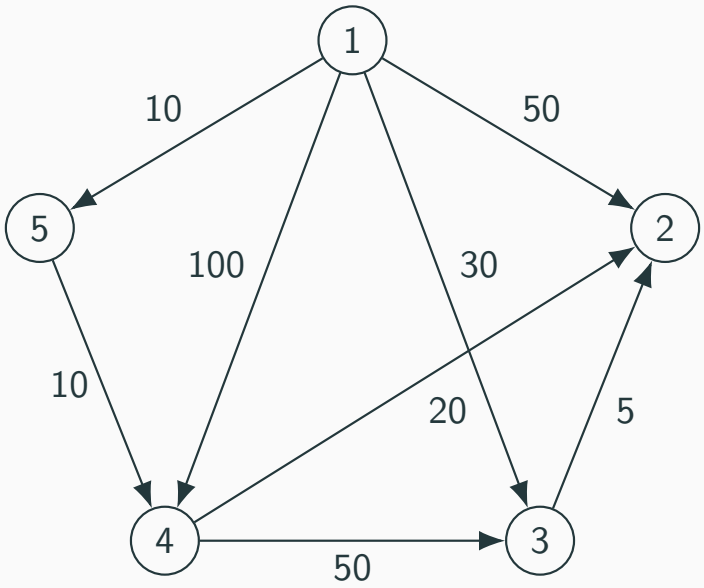
Caminos de coste mínimo

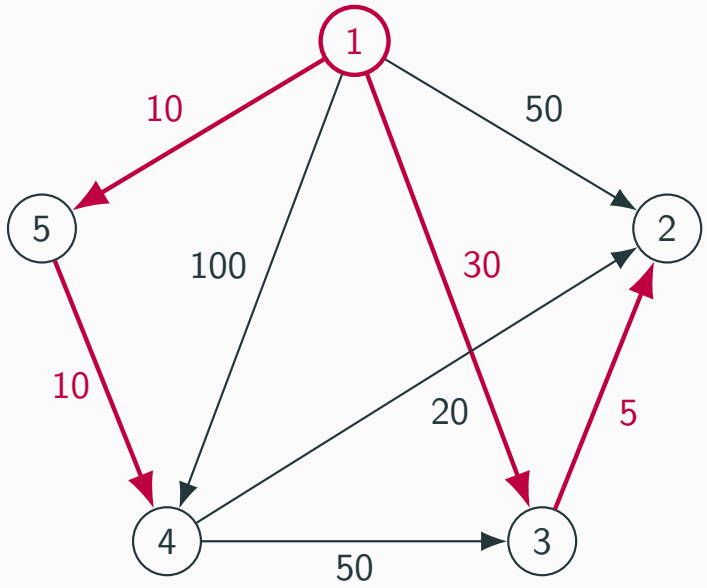
Caminos de coste mínimo

Enunciado

- Sea $G = (V, A, c)$ un grafo dirigido y ponderado, donde $c : A \to \mathbb{R}^+$ es la función que asigna a cada arista (u, v) un coste $c(u, v) > 0$.
- Uno de sus vértices, que llamaremos v_0, será el vértice origen.
- Se trata de encontrar el camino de coste mínimo desde v_0 a cualquier otro vértice v_i de G.

Caminos de coste mínimo





Objetivo

- Partir del vértice 1.
- Encontrar el camino de coste mínimo del vértice 1 al resto de vértices.

Algoritmo de Dijkstra: idea general

- S es el conjunto que contiene los vértices que ya han sido elegidos (aquellos cuyo camino de coste mínimo desde el vértice v_0 ya es conocido):
  - Inicialmente, solo contiene a v_0.
  - Al finalizar, contiene todos los vértices del grafo.
- C es el conjunto que contiene todos los demás vértices, es decir, $C = V \setminus v_0$.
- En cada paso, se selecciona el vértice de C cuyo camino al origen sea de coste mínimo, y se añade a S.

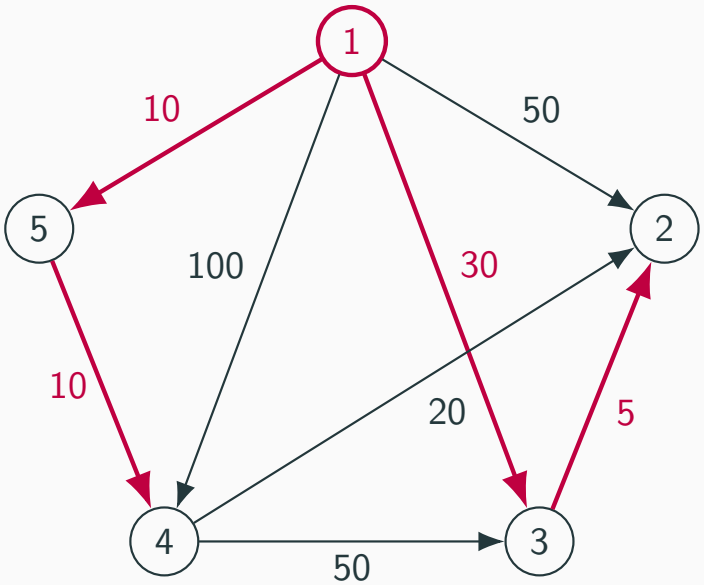
Algoritmo de Dijkstra: idea general

- Un camino desde el origen v_0 hasta otro vértice $v \in C$ es **especial** si todos los vértices intermedios del camino están en S.
- Existe un vector D que contiene el coste del camino especial de coste mínimo que va hasta cada vértice del grafo.
- Cuando se desea añadir un nuevo vértice v a S, el camino especial más corto hasta v es también el más corto de los caminos posibles hasta v. Inicialmente, $D[v_i] = c(v_0, v_i)$.
- Al finalizar el algoritmo, todos los vértices del grafo se encuentran en S (todos los caminos desde el origen hasta algún otro vértice son especiales).
- Los valores que hay en D dan la solución del problema de caminos de coste mínimo.

Algoritmo de Dijkstra: idea general

- Se supone que los vértices de G están numerados desde 1 hasta n (el vértice 1 es el v_0).
- La matriz L contiene el coste de todas las aristas dirigidas:
  - $L[i][j] > 0$, si la arista $(i, j) \in A$.
  - $L[i][j] = \infty$, en otro caso.
- Para determinar no solo el mínimo coste de los caminos sino también por dónde pasan, se añade el vector P, en donde $P[v_i]$ contiene el número del vértice que precede a v_i dentro del camino de mínimo coste. Inicialmente, $P[v_i] = v_0$.
- Para hallar el camino de mínimo coste, se siguen los punteros P hacia atrás, desde el destino hacia el origen.

Caminos de coste mínimo



Datos

- $D = [0, 35, 30, 20, 10]$.
- $P = [1, 3, 1, 5, 1]$.

¿Se puede reconstruir la ruta usando P?

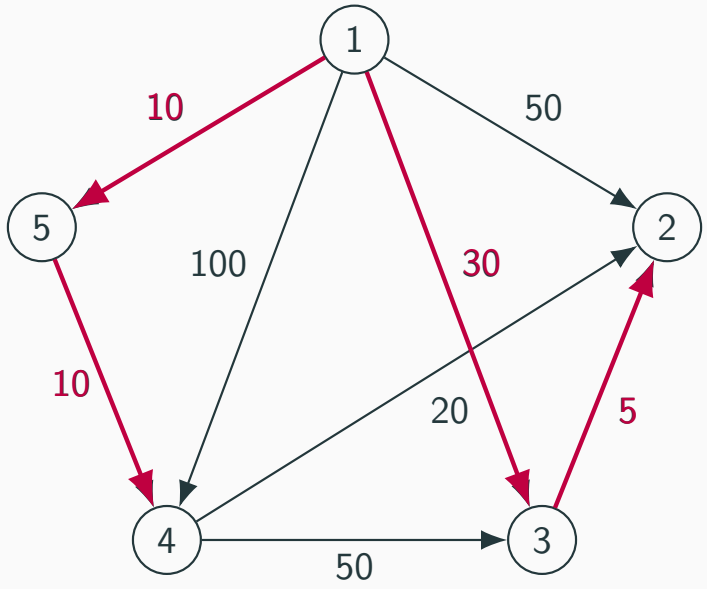
Algoritmo de Dijkstra: diseño voraz

- **Lista de candidatos**: los vértices del grafo, salvo v_0.
- **Lista de candidatos seleccionados**: los vértices que se han seleccionado para obtener las aristas que forman la solución.
- **Función solución**: se han seleccionado $n-1$ vértices del grafo.
- **Función de selección**: se selecciona el vértice v_i no utilizado cuyo $D[v_i]$ sea mínimo.
- **Función de factibilidad**: siempre es factible insertar un vértice en la solución.
- **Función objetivo**: la distancia de un vértice inicial v_0 al resto de vértices del grafo.

Algoritmo de Dijkstra: pseudocódigo

```
function Dijkstra(L[1..n][1..n], S)
    C ← {2, 3, ..., n}
    for all i ∈ [1..n] do
        D[i] ← L[1][i]
        P[i] ← 1
    end for
    while C ≠ ∅ do
        v ← aquel elemento de C que minimiza D[v]
        C ← C \ {v}
        for all w ∈ C do
            if D[w] > D[v] + L[v][w] then
                D[w] ← D[v] + L[v][w]
                P[w] ← v
            end if
        end for
    end while
    return D, P
end function
```

Caminos de coste mínimo



Paso	v	S	C	D	P
Inicio	-	$\{1\}$	$\{2,3,4,5\}$	$[0,50,30,100,10]$	$[1,1,1,1,1]$
1	5	$\{1,5\}$	$\{2,3,4\}$	$[0,50,30,20,10]$	$[1,1,1,5,1]$
2	4	$\{1,4,5\}$	$\{2,3\}$	$[0,40,30,20,10]$	$[1,4,1,5,1]$
3	3	$\{1,3,4,5\}$	$\{2\}$	$[0,35,30,20,10]$	$[1,3,1,5,1]$
3	2	$\{1,2,3,4,5\}$	$\{\emptyset\}$	$[0,35,30,20,10]$	$[1,3,1,5,1]$

Óptimo

- El algoritmo de Dijkstra halla los caminos de coste mínimo desde un vértice origen v_0 a todos los demás vértices del grafo.

Demostración

- Se demostrará por inducción que:
  1. $v_i \in S$ $(v_i \neq v_0) \Rightarrow D[i]$ es el coste del camino óptimo desde v_0 a v_i.
  2. $v_i \notin S \Rightarrow D[i]$ es el coste del camino especial óptimo desde v_0 a v_i.
- **Base**. Inicialmente, $S = \{v_0\}$. El único camino especial a cualquier vértice v_i es el directo, y D se inicia con esos costes
- **Hipótesis**. Las propiedades (1) y (2) son válidas justo antes de añadir a S el vértice v que minimiza $D[v]$ entre los vértices $v \in C$.

Caminos de coste mínimo

- Paso de inducción para la situación (1):
  - Si $v_i \in S$ antes de añadir v a S, $v_i \in S$ también después de hacerlo, por lo que (1) seguirá siendo válida.
  - No obstante, antes de añadir v a S, es preciso comprobar que $D[v]$ es el coste del camino óptimo de v_0 a v.
  - La hipótesis de inducción dice que $D[v]$ es el coste del camino especial óptimo de v_0 a v. Por tanto, basta comprobar que el camino óptimo de v_0 a v no incluye ningún vértice que no pertenezca a S.
  - Sea u el primero de esos vértices. El camino que va desde v_0 a u es un camino especial y su coste, aplicando la hipótesis de inducción, será $D[u]$.
  - Claramente, el coste del camino que va de v_0 a v a través de u es mayor o igual que $D[u]$, es decir, $D[v] \geq D[u]$, ya que los costes de los arcos son positivos.
  - Por otra parte, $D[u] \geq D[v]$, ya que $v = arg\,\min_{\forall w \in C}(D[w])$.
  - Por tanto, el coste del camino que va de v_0 a v a través de u es, como mínimo, igual a $D[v]$ y no mejora el del camino especial óptimo de v_0 a v.

Caminos de coste mínimo

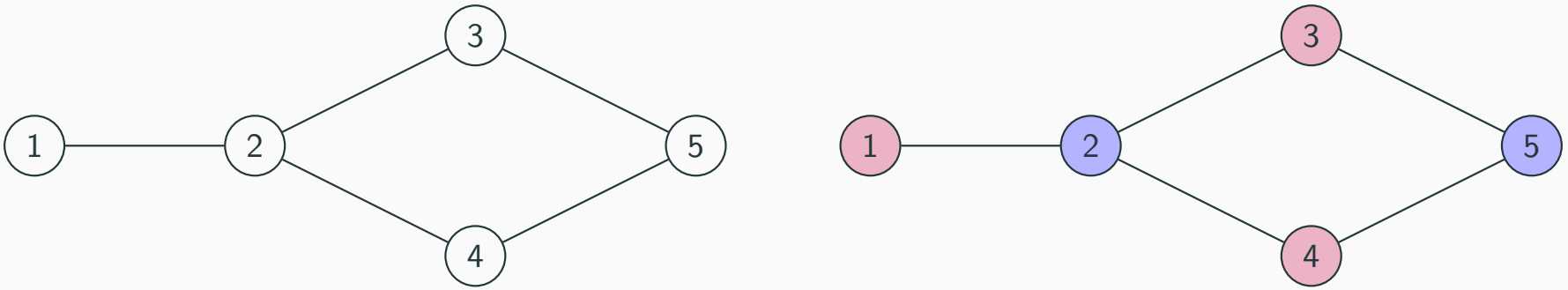
- Paso de inducción para la situación (2):
  - Considérese un vértice $u \notin S$, distinto de v. Cuando v se añade a S, hay dos opciones para el camino especial de coste mínimo desde v_0 hasta u: o bien no cambia, o bien ahora pasa a través de v (y, posiblemente, también a través de otros vértices de S).
  - En el segundo caso, sea x el último vértice de S visitado antes de llegar a u. El coste de ese camino es $D[x] + c(x, u)$.
  - A primera vista, podría parecer que para calcular el nuevo valor de $D[u]$ deberíamos comparar el valor actual de $D[u]$ con el de todos los posibles caminos a través de S: $D[x] + c(x, u)$ para todo $x \in S$. Sin embargo, esta comparación ya fue hecha en el momento de añadir x a S, para todo $x \neq v$, y $D[x]$ no ha variado desde entonces. Por tanto, basta comparar el valor actual de $D[u]$ con $D[v] + c(v, u)$ y quedarse con el mínimo.
  - Puesto que el algoritmo hace precisamente eso, la situación (2) seguirá siendo cierta después de añadir a S un nuevo vértice v.

El coloreo de un grafo

El coloreo de un grafo

Enunciado

- Sea $G = (V, A)$ un grafo no dirigido. Se desea asignar un color a cada vértice del grafo de forma que:
  - No haya dos vértices adyacentes con el mismo color.
  - Se utilice el mínimo número de colores posibles.



Teorema de los cuatro colores (Appel y Haken, 1976)

- Un grafo plano requiere a lo sumo cuatro colores para pintar sus vértices de modo que no haya vértices adyacentes con el mismo color.

El coloreo de un grafo

Complejidad

- Clase de problemas P. Aquellos que son resolubles en un tiempo igual o inferior a orden polinómico $O(n^k)$.
- Clase de problemas NP. Aquellos cuya solución es fácil de verificar, pero cuyo cálculo conlleva una complejidad muy grande debido a la gran cantidad de potenciales soluciones (óptimas y no óptimas) a explorar.
- El problema del coloreo de un grafo es NP, por lo que no se conoce una solución eficiente para resolverlo.

Heurísticas

- Al no conocerse soluciones óptimas eficientes a los problemas NP, si la eficiencia es un requisito en la implementación de la solución, se aceptan soluciones «eficientemente buenas» o «las mejores que se pueden encontrar».
- Las heurísticas son criterios, métodos, o principios para decidir cuál, entre una serie de alternativas de acción, promete ser más efectiva para lograr una meta concreta.
- Las heurísticas representan un compromiso entre dos exigencias:
  - Desde el punto de vista computacional, deben ser simples.
  - Deben discriminar correctamente entre buenas y malas opciones.

El coloreo de un grafo

Idea general

- Inicialmente, suponemos que ningún vértice del grafo está coloreado.
- Se selecciona, al azar, un vértice del grafo y se pinta de un color.
- Se escogen todos los vértices no adyacentes del vértice seleccionado que aún no estén coloreados y se pintan del mismo color.
- Se repite el procedimiento anterior hasta que todos los vértices del grafo estén pintados.

El coloreo de un grafo

Algoritmo: diseño voraz

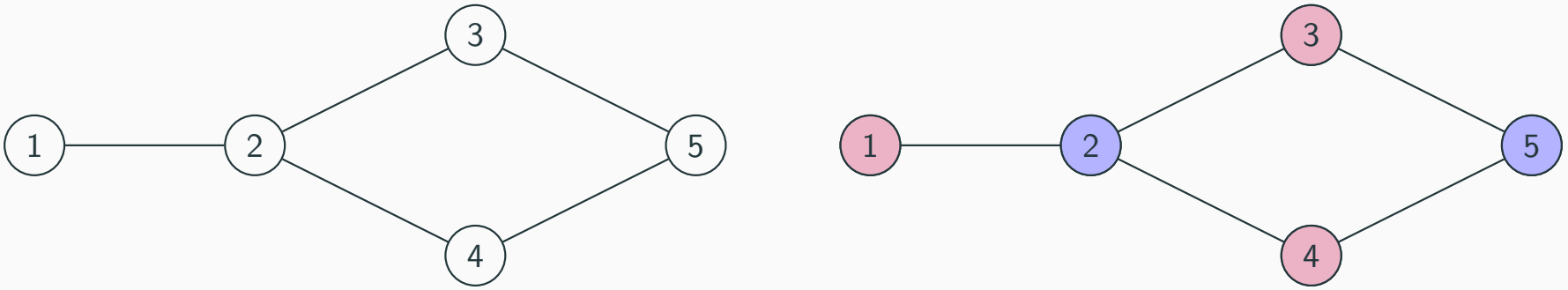
- **Lista de candidatos**: los vértices del grafo.
- **Lista de candidatos seleccionados**: los vértices ya coloreados.
- **Función solución**: todos los vértices del grafo están coloreados.
- **Función de selección**: se selecciona un vértice al azar.
- **Función de factibilidad**: el vértice seleccionado no puede estar coloreado.
- **Función objetivo**: minimizar el número de colores usados para colorear los vértices del grafo.

El coloreo de un grafo

Pseudocódigo

function Coloreo($G = (V, A)$)
    $C \leftarrow V$ // Candidatos no utilizados
    $n \leftarrow$ número de vértices de V
    $T \leftarrow \{0\}_n$ // Colores asignados a cada vértice
    $K \leftarrow 1$
    while $|C| > 0$ **do**
        $i \leftarrow random(C)$
        $C \leftarrow C \setminus \{i\}$
        $T[i] \leftarrow K$
        for all $j \in C$ tal que $\nexists (i, j) \in A$ y $\nexists (u, j) \in A$ tal que $T[u] = K$ **do**
            $T[j] \leftarrow K$
            $C \leftarrow C \setminus \{j\}$
        end for
        $K \leftarrow K + 1$
    end while
    return T
end function

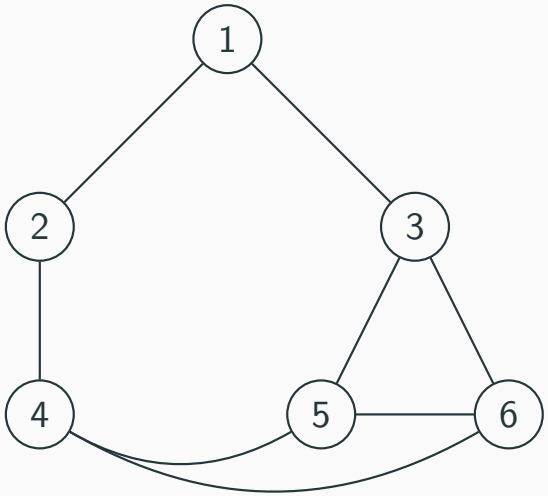
El coloreo de un grafo

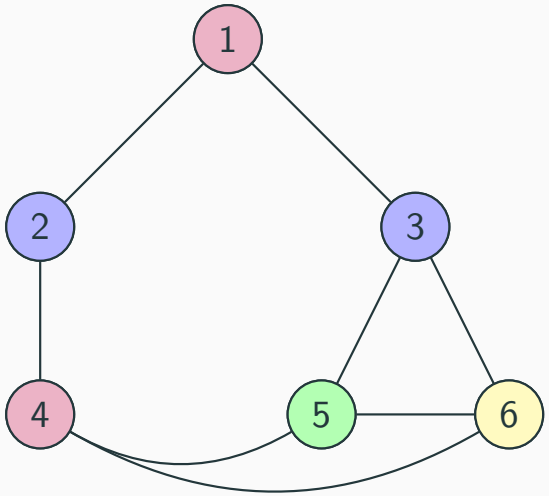


Pasos del algoritmo

1. Seleccionar el vértice 3 (al azar) y colorearlo.
2. Rellenar también 1 y 4 con el mismo color.
3. Seleccionar el vértice 2 (al azar) y colorearlo.
4. Rellenar también 5 con el mismo color.

El algoritmo voraz no es óptimo: contraejemplo

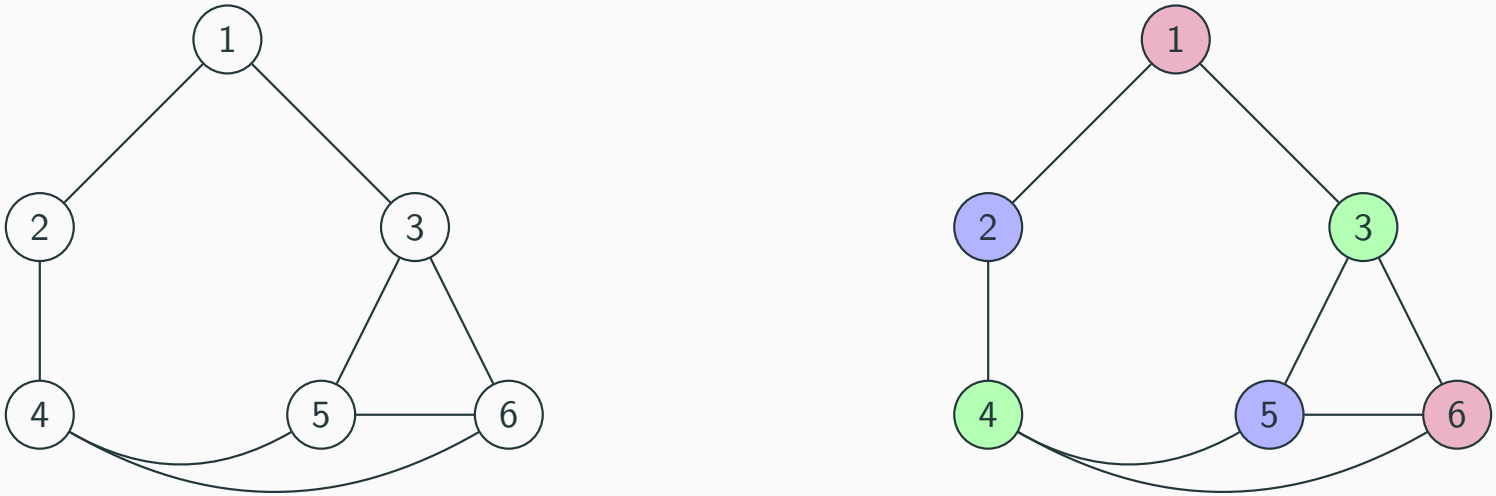




Pasos del algoritmo

1. Seleccionar el vértice 1 (al azar) y colorearlo.
2. Rellenar también 4 con el mismo color.
3. Seleccionar el vértice 2 (al azar) y colorearlo.
4. Rellenar también 3 con el mismo color.
5. Seleccionar el vértice 5 (al azar) y colorearlo.
6. Seleccionar el vértice 6 (al azar) y colorearlo.

El algoritmo voraz no es óptimo: contraejemplo



Se puede colorear con tres colores (solución óptima)

El viajante de comercio

El viajante de comercio

Enunciado

- Se conocen las distancias entre un determinado número de ciudades.
- Partiendo de una de ellas, un viajante debe visitar cada ciudad exactamente una vez y regresar al punto de partida habiendo recorrido en total la menor distancia posible.

Enunciado formal

- Dado un grafo $G = (V, A)$ no dirigido, conexo, ponderado y completo, y dado uno de sus vértices v_0, encontrar el ciclo hamiltoniano de coste mínimo que comienza y termina en v_0.

El viajante de comercio

Complejidad

- El problema del viajante de comercio es NP, por lo que no se conoce una solución eficiente para resolverlo.

Heurísticas

- Vértices como candidatos. Escoger el vértice más cercano al último vértice añadido, siempre que no se haya seleccionado previamente y que no cierre el camino antes de pasar por todos los vértices.
- Aristas como candidatos. Como en el algoritmo de Kruskal, pero garantizando que al final se forme un ciclo y que de cada vértice no salgan más de dos aristas.

El viajante de comercio

Idea general

- Inicialmente, suponemos que se parte de un vértice cualquiera del grafo.
- A continuación, se selecciona el vértice adyacente no visitado más cercano, e insertamos la arista que lo une en la solución.
- Al finalizar, se une la arista del último vértice seleccionado con el primero para cerrar el ciclo.

El viajante de comercio

Algoritmo: diseño voraz

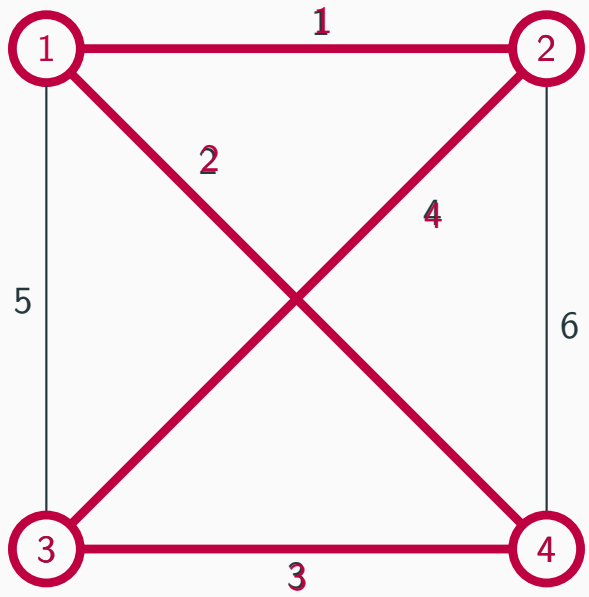
- **Lista de candidatos**: los vértices del grafo.
- **Lista de candidatos seleccionados**: los vértices ya seleccionados.
- **Función solución**: todos los vértices del grafo han sido seleccionados.
- **Función de selección**: se selecciona un vértice tal que la arista que lo une con el último vértice seleccionado tiene coste mínimo.
- **Función de factibilidad**: no se pueden formar ciclos (excepto al añadir el último vértice).
- **Función objetivo**: encontrar un ciclo hamiltoniano de coste mínimo en el grafo.

El viajante de comercio

Peudocódigo

function ViajanteDeComercio($G = (V, A)$)
    $s \leftarrow random(V)$ // Vértice cualquiera
    $C \leftarrow V \setminus \{s\}$
    $T \leftarrow \{\emptyset\}$
    while $|C| > 0$ **do**
        buscar $a = (s, v)$ de coste mínimo tal que $v \in C$
        $C \leftarrow C \setminus \{v\}$
        $T \leftarrow T \cup \{a\}$
        $s \leftarrow v$
    end while
    $a \leftarrow (s, T[1])$
    $T \leftarrow T \cup \{a\}$
    return T
end function

El viajante de comercio

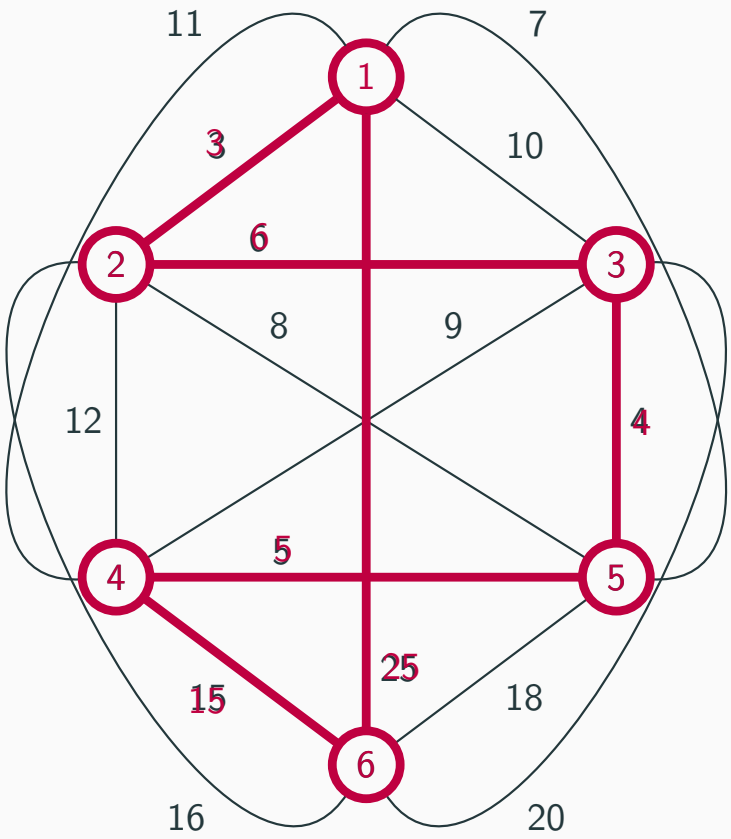


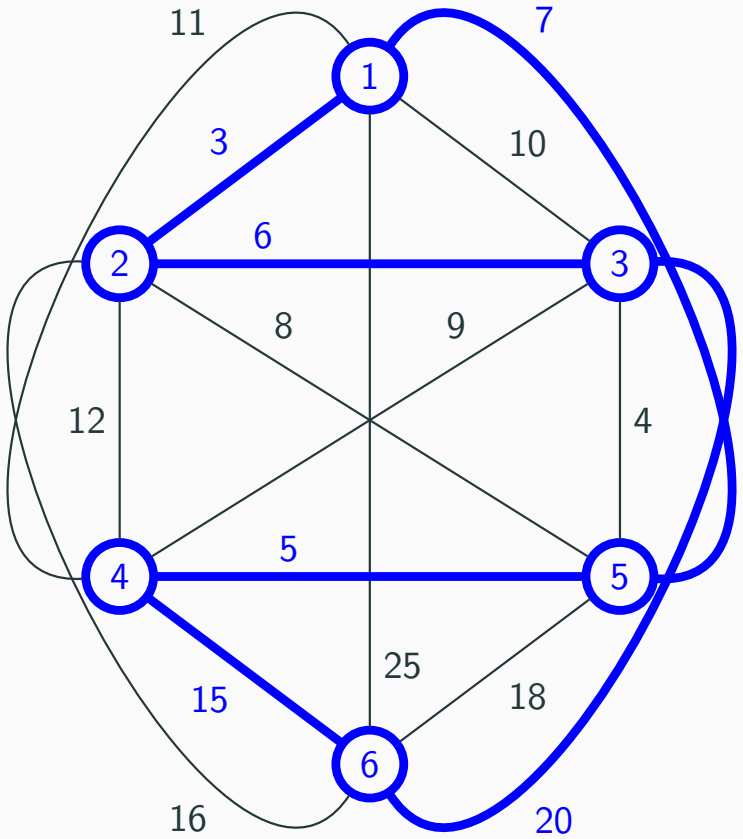
Paso	v	s	C	Arista	T
Inicio	-	1	$\{2,3,4\}$	-	-
1	2	1	$\{3,4\}$	$(1,2)$	$(1,2)$
2	3	2	$\{4\}$	$(2,3)$	$(1,2),(2,3)$
3	4	3	$\{\emptyset\}$	$(3,4)$	$(1,2),(2,3),(3,4)$
4	1	4	$\{\emptyset\}$	$(4,1)$	$(1,2),(2,3),(3,4),(4,1)$

Solución óptima. Coste: $1+4+3+2=10$

El viajante de comercio

El algoritmo voraz no es óptimo: contraejemplo





- Coste de la solución del algoritmo: $3 + 6 + 4 + 5 + 15 + 25 = 58$
- Coste de la solución óptima: $3 + 6 + 20 + 15 + 5 + 7 = 56$

Consideraciones finales

Consideraciones finales

¿Por qué no emplear siempre algoritmos voraces?

- No todos los problemas admiten esta estrategia de resolución.
- La búsqueda de óptimos locales no tiene por qué conducir a un óptimo global.

Estrategia voraz: limitaciones

- Trata de ganar todas las batallas sin ser consciente de que para ganar la guerra muchas veces se necesita perder alguna batalla.
- Lo que parece bueno hoy no siempre tiene por qué ser bueno en el futuro.

Consideraciones finales

- Es fundamental encontrar la función de selección adecuada que garantice que el candidato escogido o rechazado es el que debe formar parte o no de la solución óptima sin posibilidad de reconsiderar tal decisión.
- Al plantear un algoritmo voraz (que por diseño será eficiente), hay que demostrar formalmente que conduce a la solución óptima del problema en todos los casos.
- En el caso de que no conduzca a soluciones óptimas, hay que presentar un contraejemplo para el cual el algoritmo no proporcione la solución óptima.

Consideraciones finales

- Debido a su eficiencia, los algoritmos voraces se usan incluso cuando no encuentran la solución óptima.
- A veces, es imprescindible encontrar pronto, o en un tiempo finito, una solución razonablemente buena, aunque no sea la óptima.
- Otras veces, un algoritmo voraz da una solución rápida (no óptima) del problema y, a partir de ella, aplicando algoritmos más sofisticados, la solución óptima se obtiene más rápidamente.
- Los algoritmos voraces se aplican en muchas circunstancias debido a su eficiencia, aunque solo den una solución aproximada al problema de optimización planteado.

Bibliografía

Bibliografía

Aclaración

- El contenido de las diapositivas es esquemático y representa un apoyo para las clases teóricas.
- Se recomienda completar los contenidos del tema 1 con apuntes propios tomados en clase y con la bibliografía principal de la asignatura.

Por ejemplo


G. Brassard and P. Bratley.
***Fundamentals of Algorithmics*.**
Prentice Hall, Englewwod Cliffs, New Jersey, 1996.


J. L. Verdegay.
***Lecciones de Algorítmica*.**
Editorial Técnica AVICAM, 2017.