TEMA 3

Greedy

Esquema general Greedy (método voraz)

- Características generales de los algoritmos voraces (I):
  - Para construir la solución se dispone de un conjunto de candidatos. A medida que avanza el algoritmo se van formando dos conjuntos: el **conjunto de candidatos considerados y seleccionados** (formarán parte de la solución) y el **conjunto de candidatos considerados y rechazados** definitivamente
  - Existe una **función de selección** que indica cuál es el candidato más prometedor de entre los aún no considerados
  - Existe un **test de factibilidad** que comprueba si un candidato es compatible con la solución parcial construida hasta el momento, esto es, si existe una solución incluyendo dicha solución parcial y el citado candidato

Esquema general Greedy (método voraz)

- Características generales de los algoritmos voraces (II):
  - Existe un **test de solución** que determina si una solución parcial forma una *solución completa*
  - Con frecuencia, se trata de un *problema de optimización*, es decir, se tiene que obtener una solución óptima según una **función objetivo** que asocia un valor a cada solución
- Concepto (I):
  - Los **algoritmos** voraces, ávidos o **greedy** se utilizan en problemas de optimización: es necesario realizar una serie de selecciones de manera que se minimice una función de costo
  - El **algoritmo greedy** va seleccionando una serie de elementos (usando una función de evaluación): una vez hecha la elección, no puede deshacerse, aunque después se demuestre que fue una mala elección

Esquema general Greedy (método voraz)

- Concepto (II):
  - La solución del **algoritmo greedy** está formada por un conjunto de elementos seleccionados de un conjunto de candidatos (con un orden determinado o no)
  - El **algoritmo greedy** no siempre encuentran la solución óptima
- Ejemplos:
  - Encontrar la secuencia óptima para procesar un conjunto de tareas por un computador
  - Hallar un camino de costo mínimo en un grafo
  - Hallar el recubrimiento de coste mínimo de un grafo
  - Problema del cambio de monedas. Devolver una cantidad de dinero haciendo uso del menor número de monedas

Esquema general Greedy (método voraz)

- El **algoritmo greedy** funciona por pasos:
  - Partimos de una solución vacía
  - En cada paso se escoge el siguiente elemento a añadir a la solución, entre los candidatos ⇒ Una vez hecha la elección no se podrá deshacer (no hay vuelta atrás)
  - El algoritmo acaba cuando el conjunto de elementos seleccionados constituya una solución
- Conjuntos:
  - **C**: Conjunto de elementos candidatos, pendientes de seleccionar (inicialmente todos)
  - **S**: Candidatos seleccionados para la solución
  - **R**: Candidatos que han sido seleccionados pero rechazados

Esquema general Greedy (método voraz)

- Funciones:
  - **solucion(S)**: Comprueba si un conjunto de candidatos es una solución (independientemente de que sea óptima o no)
  - **seleccionar(C)**: Devuelve el elemento más *prometedor* del conjunto de candidatos pendientes (no seleccionados ni rechazados)
  - **factible(S, x)**: Indica si a partir del conjunto S y añadiendo {x}, es posible construir una solución (posiblemente añadiendo otros elementos)
  - **funcion_objetivo(S)**: Dada una solución devuelve el coste asociado a la misma (resultado del problema de optimización)

Esquema general Greedy (método voraz)

- **Esquema algorítmico general de los métodos voraces**

```
funcion voraz(C: CjtoCandidatos) : S CjtoSolucion
    S = Ø
    mientras (C ≠ ∅) Y NO solucion(S)
            x = seleccionar(C)
            C = C - {x}
            si factible(S, x)
                    S = S ∪ {x}
    si NO solucion(S)
            devolver "No se puede encontrar solución"
    sino
            devolver S
end
```

Esquema general Greedy (método voraz)

- En general, para resolver el problema buscamos un **conjunto de candidatos** que constituya una **solución** y que optimice el valor de la **función objetivo**
- El algoritmo funciona de la siguiente manera (I):
  - Inicialmente el conjunto de candidatos escogidos (S) es vacío
  - En cada paso, se intenta añadir al conjunto **S** *el mejor* de los no escogidos, utilizando una **función de selección** basada en algún criterio de optimización
  - Tras cada paso, hay que ver si el conjunto seleccionado es **factible** (si añadiendo más candidatos se puede llegar a una **solución**)

Esquema general Greedy (método voraz)

- El algoritmo funciona de la siguiente manera (II):
  - Si el conjunto **no es factible**, se rechaza el último candidato elegido y no se vuelve a considerar
  - Si **es factible**, se **inserta** en el conjunto S y permanece siempre en él ⇒ se comprueba si el conjunto resultante es una **solución**
  - El algoritmo termina cuando se obtiene una solución
  - El algoritmo es correcto si la solución encontrada es siempre óptima

Ejemplo sencillo. Problema del cambio

- **Objetivo**: Se trata de devolver una cantidad de dinero con el menor número posible de monedas
- Se parte de:
  - Un conjunto de tipos de monedas válidas (C), de las que se supone que hay cantidad suficiente para realizar el desglose, y
  - Una cantidad a devolver (*cant*)
- Elementos fundamentales del esquema voraz (I):
  - **Conjunto de candidatos** (C): cada una de las monedas de los diferentes tipos

    C = {2€, 1€, 0.50€, 0.20€, 0.10€, 0.05€, 0.02€, 0.01€}

Ejemplo sencillo. Problema del cambio

- Elementos fundamentales del esquema voraz (II):
  - **Solución** (S): conjunto de monedas devuelto y cuyo valor total es igual a la cantidad a devolver. $S = \{x_i : \sum_i x_i * v_i = cant\}$
  - **Factible**: la suma de los valores de las monedas escogidas no supera la cantidad a devolver $\sum_i x_i * v_i \leq cant$
  - **Función de selección**: elegir la moneda de mayor valor de entre las candidatas, sin sobrepasar la cantidad que aún queda por devolver (cant')
    $v_i = \max\{v_j\} : v_j = \{v_k \in C : v_k < cant'\}$
  - **Función objetivo**: minimizar número total de monedas utilizadas
    $f(x_1, x_2, \ldots, x_n) = x_1 + x_2 + \cdots + x_n$
    verificándose que: $\sum_i x_i * v_i = cant$

Ejemplo sencillo. Problema del cambio

- **Esquema algorítmico general de los métodos voraces**

```
funcion devolverCambio(C: conjCand, cant): S Sol
    C = {2, 1, 0.50, 0.20, 0.10, 0.05, 0.02, 0.01}
    S = ∅
    suma = 0 // suma parcial de S
    mientras suma ≠ cant
            x = mayorElemento(C) tal que suma + x ≤ cant
            si no existe x
                    devolver "No hay solución"
            C = C - {x}  // se supone que hay suficientes monedas de cada
    tipo
            S = S ∪ {x}
            suma = suma + x
    devolver S
end
```

Algoritmos Greedy en grafos. Caminos mínimos (**Dijkstra**)

- **Objetivo**: dado un grafo dirigido, **G = (V, E)**, con una función de peso asociada a sus aristas (*pesos no negativos*), tomando un vértice como **fuente**, obtener el camino de coste mínimo que va desde ese vértice a cada uno de los restantes vértices de V
- Solución voraz (I):
  - **Conjunto de candidatos**: Vértices del grafo para los que hay que resolver el problema (inicialmente todos menos el propio fuente)
  - **Solución parcial S**: Vértices a los cuales ya sabemos llegar usando el camino mínimo desde el vértice fuente (inicialmente ∅)
  - **Función de selección**: hallar el vértice **w** del conjunto de candidatos (V - S) que está a menor distancia del vértice fuente

Algoritmos Greedy en grafos. Caminos mínimos (**Dijkstra**)

- Solución voraz (II):
  - **Función de factibilidad**: el nuevo vértice candidato es siempre factible si su distancia al vértice fuente es la mínima
  - **Función objetivo**: caminos mínimos, es decir, con la menor distancia (coste), desde fuente a cada vértice restante del grafo
  - **Función de solución**: no se utiliza
- Algoritmo estudiado e implementado en EDA I

  D ⇒ mapa de costes de caminos mínimos, almacenando la distancia del vértice fuente a cada uno de los otros vértices del grafo. D[i] almacena la menor distancia entre el vértice fuente y el vértice i

  S ⇒ mapa formado por los vértices de los cuales ya

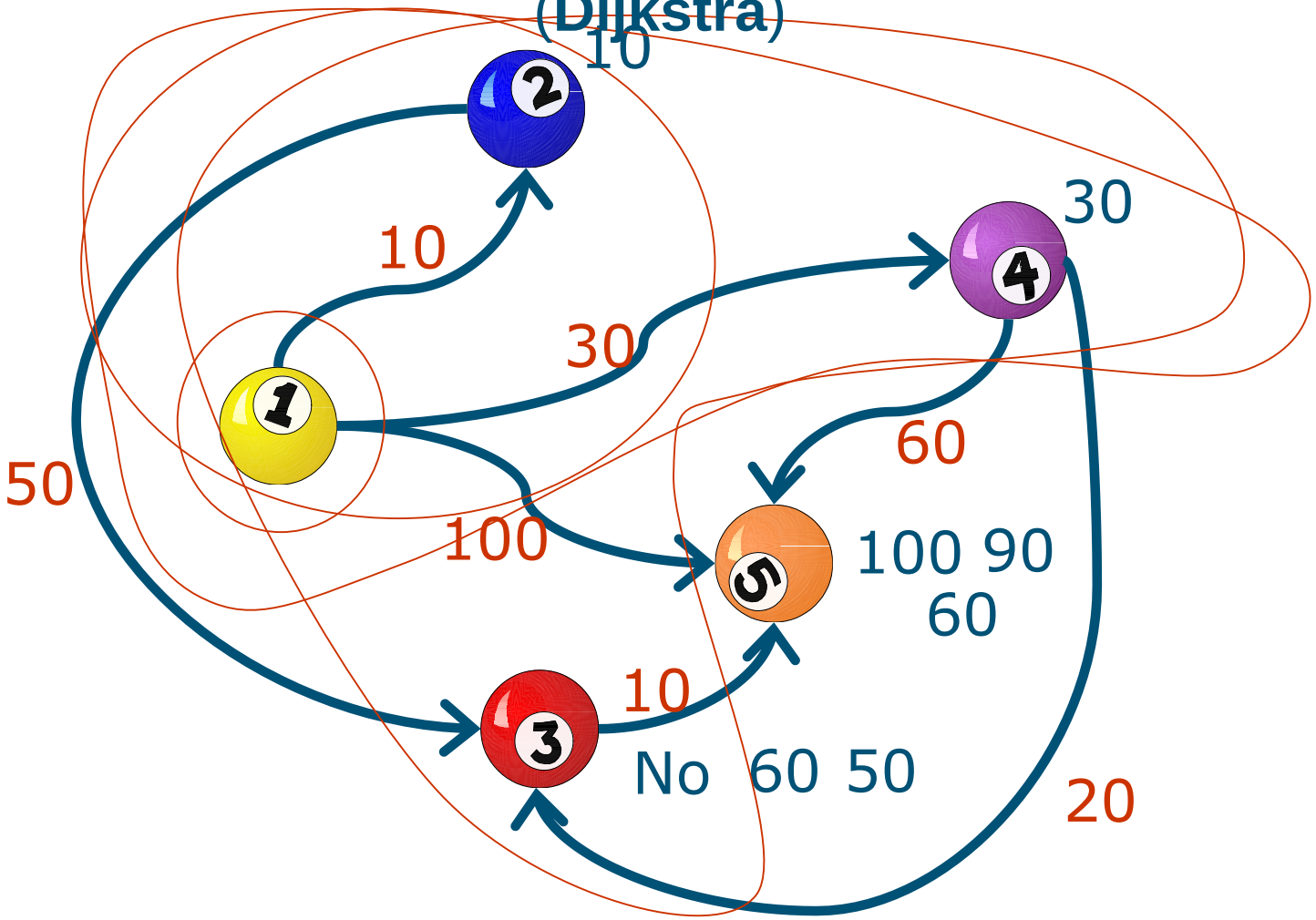
Algoritmos Greedy en grafos. Caminos mínimos (**Dijkstra**)

```
funcion Dijkstra(Grafo G, fuente)
    TreeMap<Vertex, Double> D
    TreeMap<Vertex, Vertex> S
    TreeSet<Vertex> V_menos_S
    Inicializar V_menos_S con todos los vértices de V menos fuente
    Inicializar D y S para cada v ∈ V_menos_S
        D[v] = (adyacentes(fuente, v)) ? peso(fuente, v) : ∞
        S[v] = (adyacentes(fuente, v)) ? fuente : null
    mientras V_menos_S no esté vacío
        Encontrar el vértice u ∈ V_menos_S tal que D[u] sea el valor mínimo
        Eliminar u de V_menos_S
        para cada vértice v de V_menos_S
            si (adyacentes(u, v))
                si (D[u] + weight(u, v) < D[v])
                    D[v] = min{D[v], D[u] + weight(u, v)}
                    S[v] = u
    devolver D y S
```

Algoritmos Greedy en grafos. Caminos mínimos (**Dijkstra**)

- Tiempo de ejecución:
  - **Fase de inicialización**: O(n)
  - Bucle principal **mientras** se ejecuta n - 1 veces
    - Selección de v, requiere examinar todos los elementos de V_menos_S: $O(n^2)$
    - n eliminaciones en V_menos_S: O(n)
    - Bucle **para** interno, representa la fase de actualización de las distancias mínimas: $O(n^2)$
  - $T_{Dijkstra}(n, m) \in O(n^2)$
- Mejora: Sustituir V_menos_S por una cola de prioridad, en base a la mínima distancia al vértice fuente. Problema: la fase de actualización puede requerir cambiar la prioridad

Algoritmos Greedy en grafos. Caminos mínimos (**Dijkstra**)



Algoritmos Greedy en grafos. Caminos mínimos (**Dijkstra**)

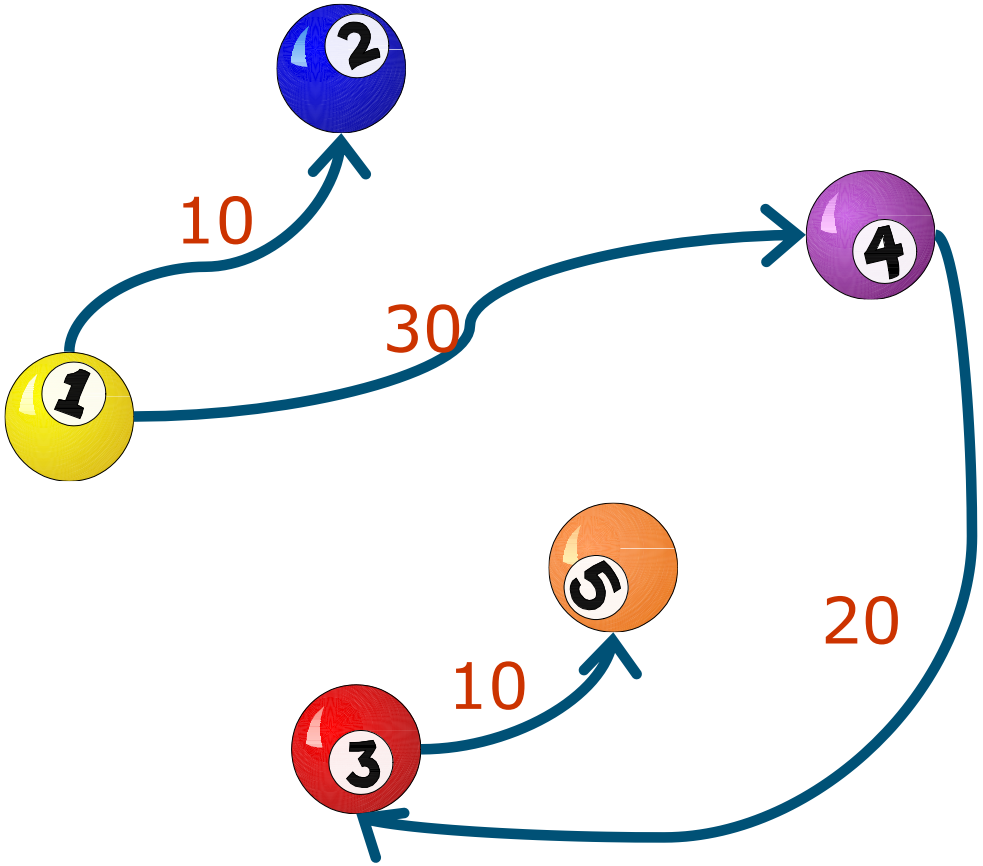
Algoritmo de Dijkstra (traza de la ejecución)

Iter	Tratado	VCM	Al vértice 2		Al vértice 3		Al vértice 4		Al vértice 5	
			D	S	D	S	D	S	D	S
Inicial	{1}		10	1	∞	1	30	1	100	1
1	{1,2}	2	10	1	60	2	30	1	100	1
2	{1,2,4}	4	10	1	50	4	30	1	90	4
3	{1,2,4,3}	3	10	1	50	4	30	1	60	3
4	{1,2,4,3,5}	5	10	1	50	4	30	1	60	3

V	D	S
1	0	1
2	10	1
3	50	4
4	30	1
5	60	3

Algoritmos Greedy en grafos. Caminos mínimos (**Dijkstra**)

Árbol para el camino de coste mínimo del vértice 1 al resto



Algoritmos Greedy en grafos. ARCM (**Prim** y **Kruscal**)

- **Objetivo** (Árbol de Recubrimiento de Coste Mínimo, **ARCM**): dado un grafo conexo no dirigido, **G = (V, E)**, con una función de peso asociada a sus aristas, obtener un subgrafo **G' = (V, T)** que solo contenga las aristas imprescindibles y para las que la suma de sus pesos sea mínima
- El subgrafo **G' = (V, T)** formado por los vértices de G y las aristas de T es un **árbol**
  - Si V contiene n vértices, entonces para que T sea óptimo, debe contener **n - 1 aristas**
  - y como G' = (V, T) es conexo, tiene que ser un **árbol**
- El grafo G' = (V,T) se llama **árbol de recubrimiento mínimo**, o **árbol generador minimal**, del grafo G

Algoritmos Greedy en grafos. ARCM (**Prim** y **Kruscal**)

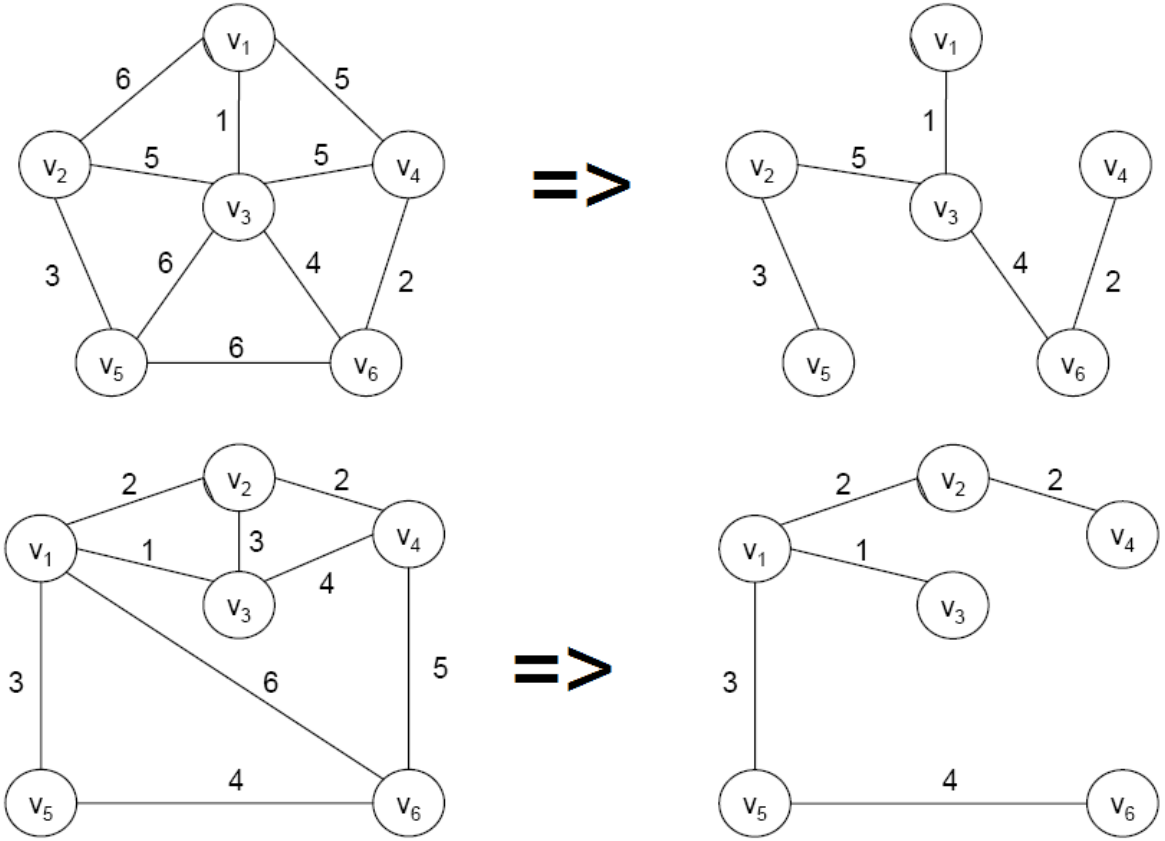
- **Aplicaciones** (**Árbol de Recubrimiento de Coste Mínimo**): Problemas relacionados con distribuciones geográficas, como el tendido de líneas de teléfono que permita intercomunicar un conjunto de ciudades, o el diseño de redes (eléctricas, hidráulicas, de ordenadores, de carreteras, etc.), minimizando el coste/precio y hallando enrutamientos sin ciclos
- **Árbol libre** es un grafo no dirigido conexo acíclico
  - Todo árbol libre con n vértices tiene n-1 aristas
  - Si se añade una arista se introduce un ciclo
  - Si se borra una arista el grafo no es conexo
  - Cualquier par de vértices está unido por un único camino simple
  - Un árbol libre con un vértice distinguido es un árbol con raíz

Algoritmos Greedy en grafos. ARCM (**Prim** y **Kruscal**)

- **Árbol de recubrimiento** de un grafo no dirigido y etiquetado no negativamente, es cualquier subgrafo que contenga todos los nodos y que sea un árbol libre
- **Árbol de recubrimiento de coste mínimo** (**ARCM**, Minimum Spanning Tree, MST) es un árbol de recubrimiento y no hay ningún otro árbol de recubrimiento con suma de aristas menor
- **Propiedad fundamental de los ARCMs**. Sea G = (V, E) un grafo no dirigido conexo y ponderado no negativamente
  - Sea S un subconjunto de vértices, tal que $S \subset V$, $S \neq \emptyset$
  - Si (u, v) es la arista de menor peso de E tal que $u \in S$ y $v \in V-S$, entonces existe algún árbol de recubrimiento de coste mínimo de G que contiene a dicha arista

Algoritmos Greedy en grafos. ARCM (**Prim** y **Kruscal**)

- **Ejemplos de árboles de recubrimiento de coste mínir**



Algoritmos Greedy en grafos. ARCM (**Prim** y **Kruscal**)

- Existen al menos **dos algoritmos voraces muy conocidos**, que resuelven este problema. En ambos se va construyendo el árbol por etapas, y en cada una se añade una arista. La forma en la que se realiza esa elección es la que los diferencia
- **Prim** $\Rightarrow$ Se parte de un vértice cualquiera (vértice *fuente*) y escoge en cada etapa la arista de menor peso que verifique que uno de sus vértices se encuentre en el conjunto de vértices ya conectados y el otro no
- **Kruskal** $\Rightarrow$ Se parte de un bosque de árboles formados por un único vértice cada uno, de forma que en cada iteración se conectan un par de árboles mediante una arista, hasta que en el bosque quede como un único árbol

Algoritmos Greedy en grafos. ARCM (**Prim** y **Kruscal**)

- **Función objetivo a minimizar**: longitud (coste) del árbol de recubrimiento
- **Candidatos**: las aristas del grafo
- **Función solución**: el conjunto de aristas seleccionado es árbol de recubrimiento de longitud mínima
- **Función factible**: el conjunto de aristas no contiene ciclos
- **Función de selección**:
  - **Prim** ⇒ Seleccionar la arista de menor peso que aún no ha sido seleccionada y que forme un árbol junto con el resto de las aristas seleccionadas
  - **Kruskal** ⇒ Seleccionar la arista de menor peso que aún no ha sido seleccionada y que no forme un ciclo

Algoritmos Greedy en grafos. ARCM (**Prim**)

- Dado el grafo G = (V, E), se mantienen dos particiones:
  - S $\Rightarrow$ vértices que se encuentran enlazados por las aristas del árbol que se está construyendo
  - V – S $\Rightarrow$ resto de vértices
- En cada iteración se incrementa el subconjunto S con un nuevo vértice, hasta que S = V
- Inicialmente, S contiene un único vértice (vértice *fuente*), que puede ser cualquiera
- En cada paso, se localiza la arista (u, v) de menor coste tal que u $\in$ S y v $\in$ V – S

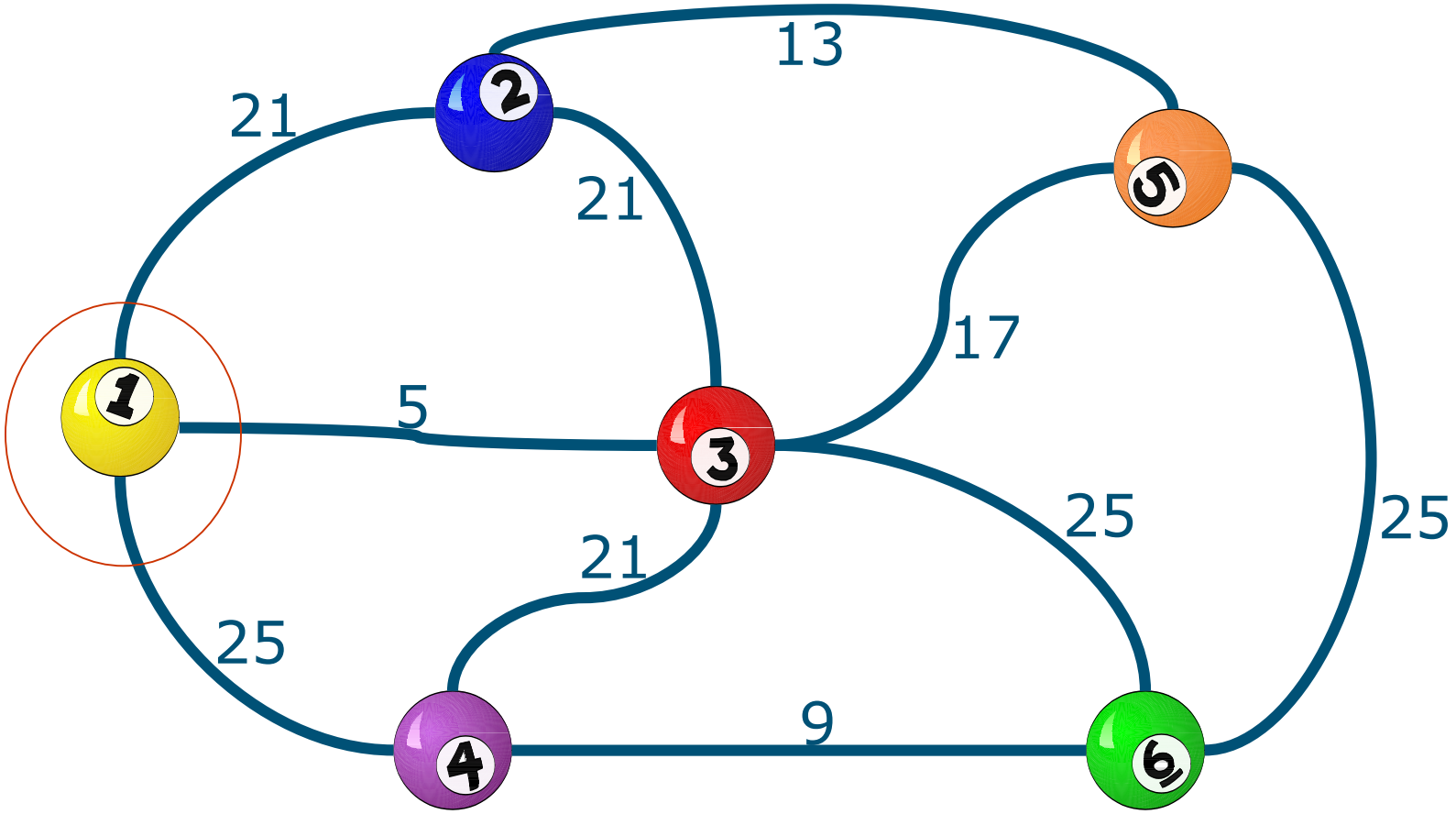
Algoritmos Greedy en grafos. ARCM (**Prim**)

```
funcion Prim(Grafo G, fuente)
    TreeMap<Vertex, Double> D   // mapa de pesos, en D[v] está el peso de la arista final (u, v)
    TreeMap<Vertex, Vertex> S   // mapa formado por los vértices u de la arista (u, v)
    ArrayList<Object> MST       // se almacenan las aristas con peso que forman el MST
    TreeSet<Vertex> V_menos_S
    Inicializar V_menos_S con todos los vértices de V menos fuente
    Inicializar D y S para cada v ∈ V_menos_S
        D[v] = (adyacentes(fuente, v)) ? peso(fuente, v) : ∞
        S[v] = (adyacentes(fuente, v)) ? fuente : null
    mientras V_menos_S no esté vacío
        Encontrar el vértice u ∈ V_menos_S tal que D[u] sea el valor mínimo
        si (se ha encontrado u)
            Eliminar u de V_menos_S
            MST = MST ∪ arista(S[u], u, peso(S[u], u))
            para cada vértice v de V_menos_S
                si (adyacentes(u, v))
                    si (weight(u, v) < D[v])
                        D[v] = weight(u, v)
                        S[v] = u
        si no break
    devolver MST
```

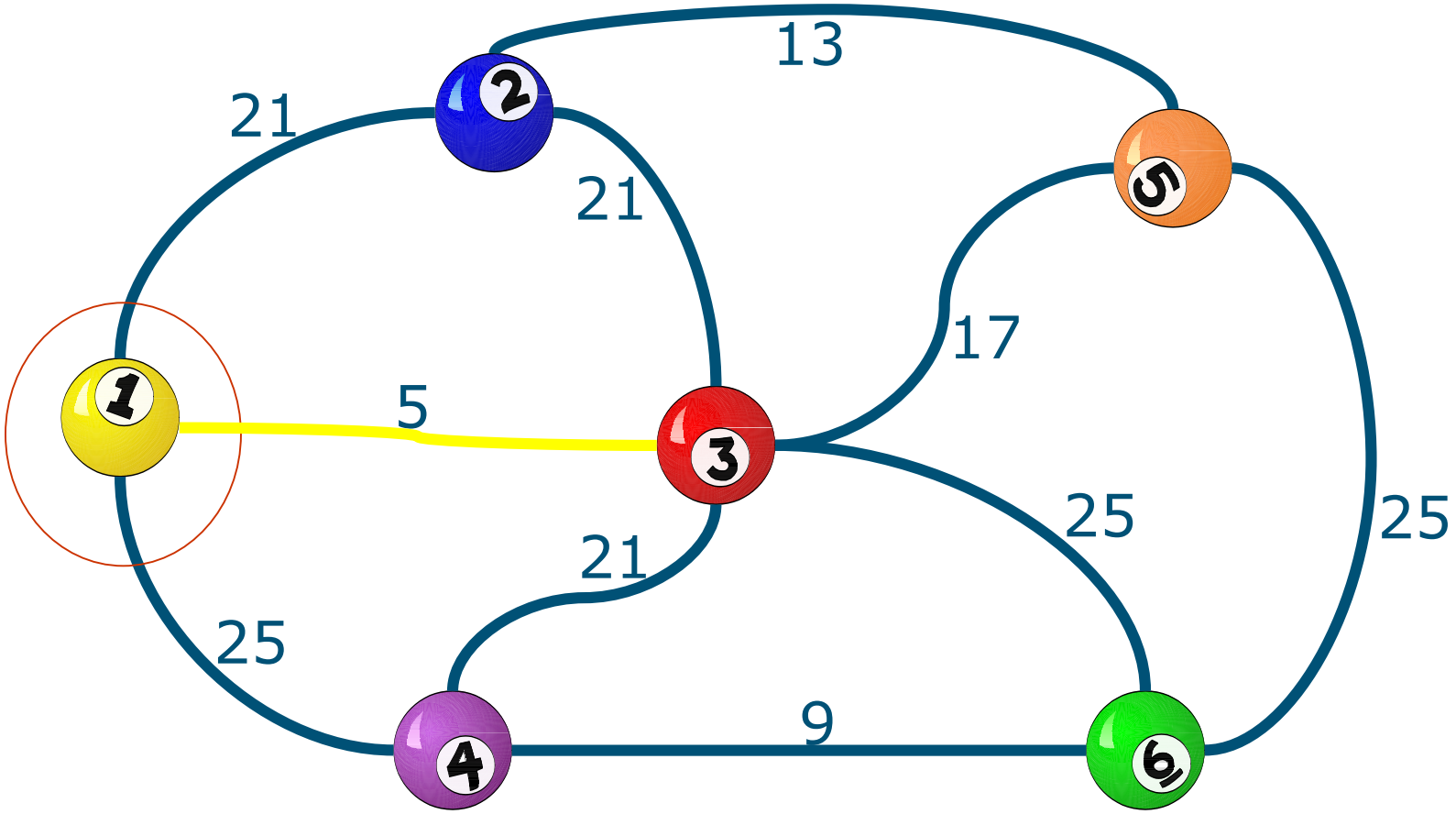
Algoritmos Greedy en grafos. ARCM (**Prim**)

- Tiempo de ejecución:
  - **Fase de inicialización**: O(n)
  - Bucle principal **mientras** se ejecuta n - 1 veces
    - Selección de v, requiere examinar todos los elementos de V_menos_S: $O(n^2)$
    - n eliminaciones en V_menos_S: O(n)
    - Bucle **para** interno, representa la fase de actualización de las estructuras D y S: $O(n^2)$
  - $T_{Prim}(n, m) \in O(n^2)$
- Mejora: En lugar de utilizar D y S, se puede utilizar una cola de prioridad de aristas con peso, ordenadas por el peso

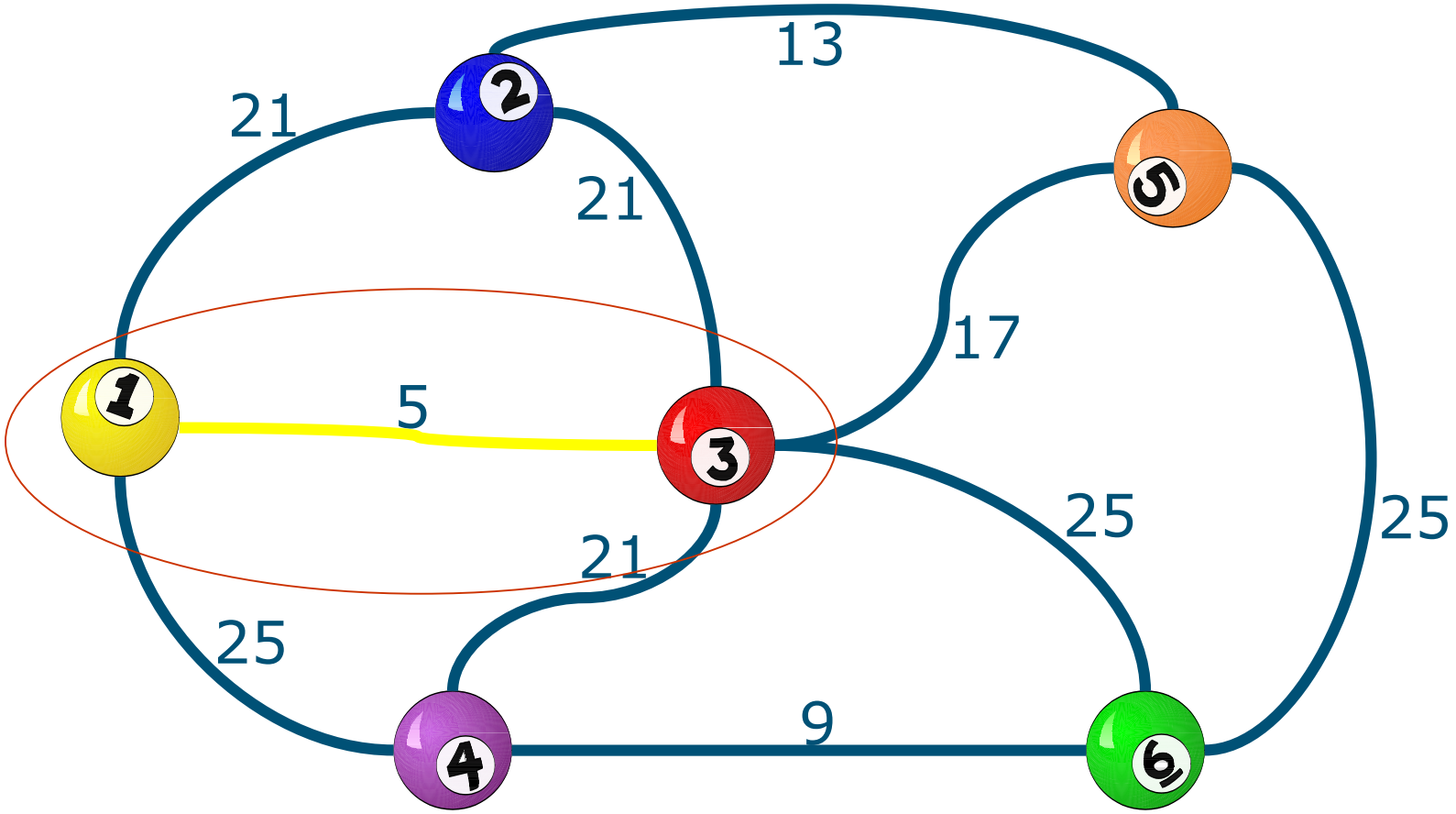
Algoritmos Greedy en grafos. ARCM (**Prim**)



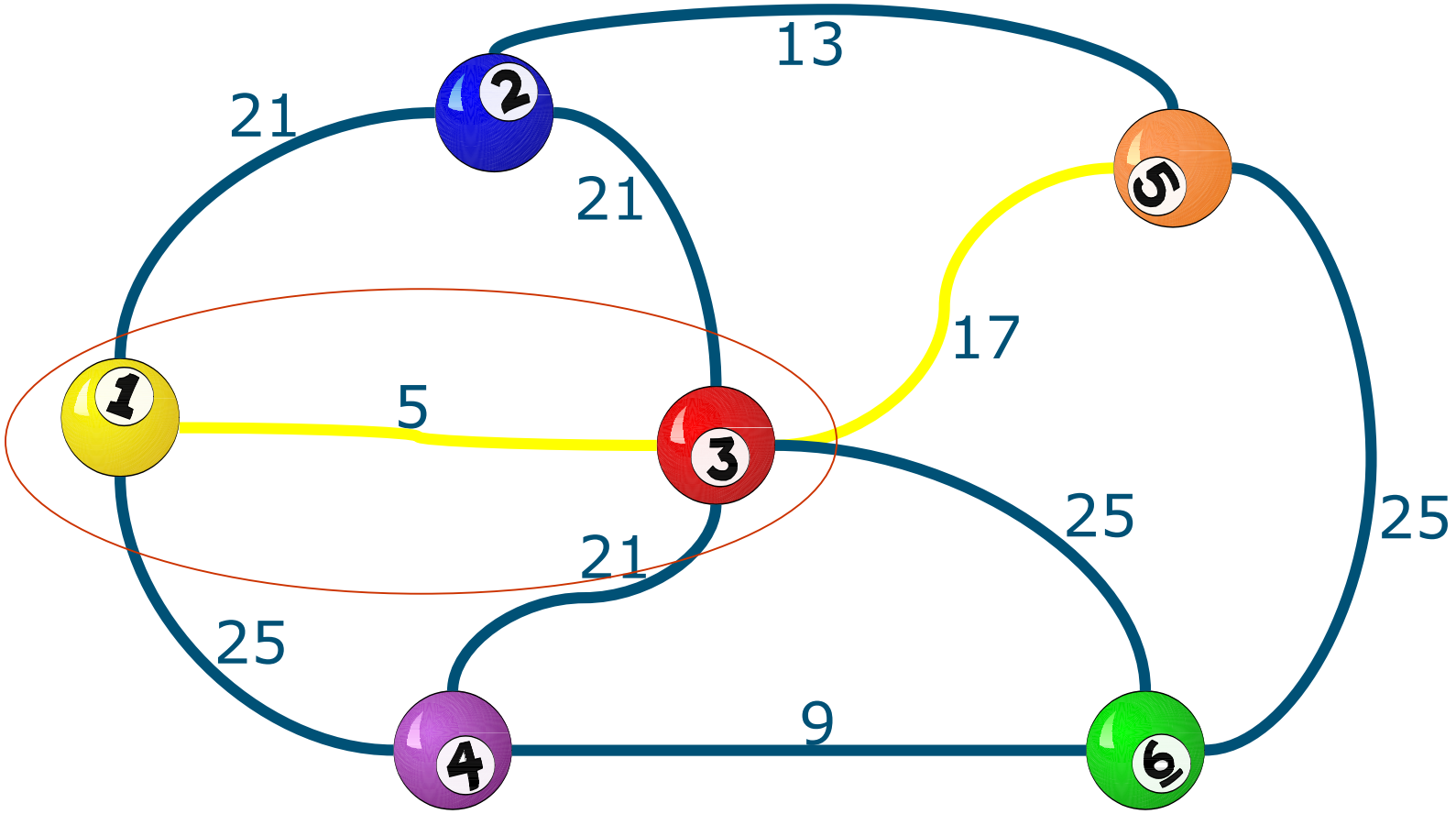
Algoritmos Greedy en grafos. ARCM (**Prim**)



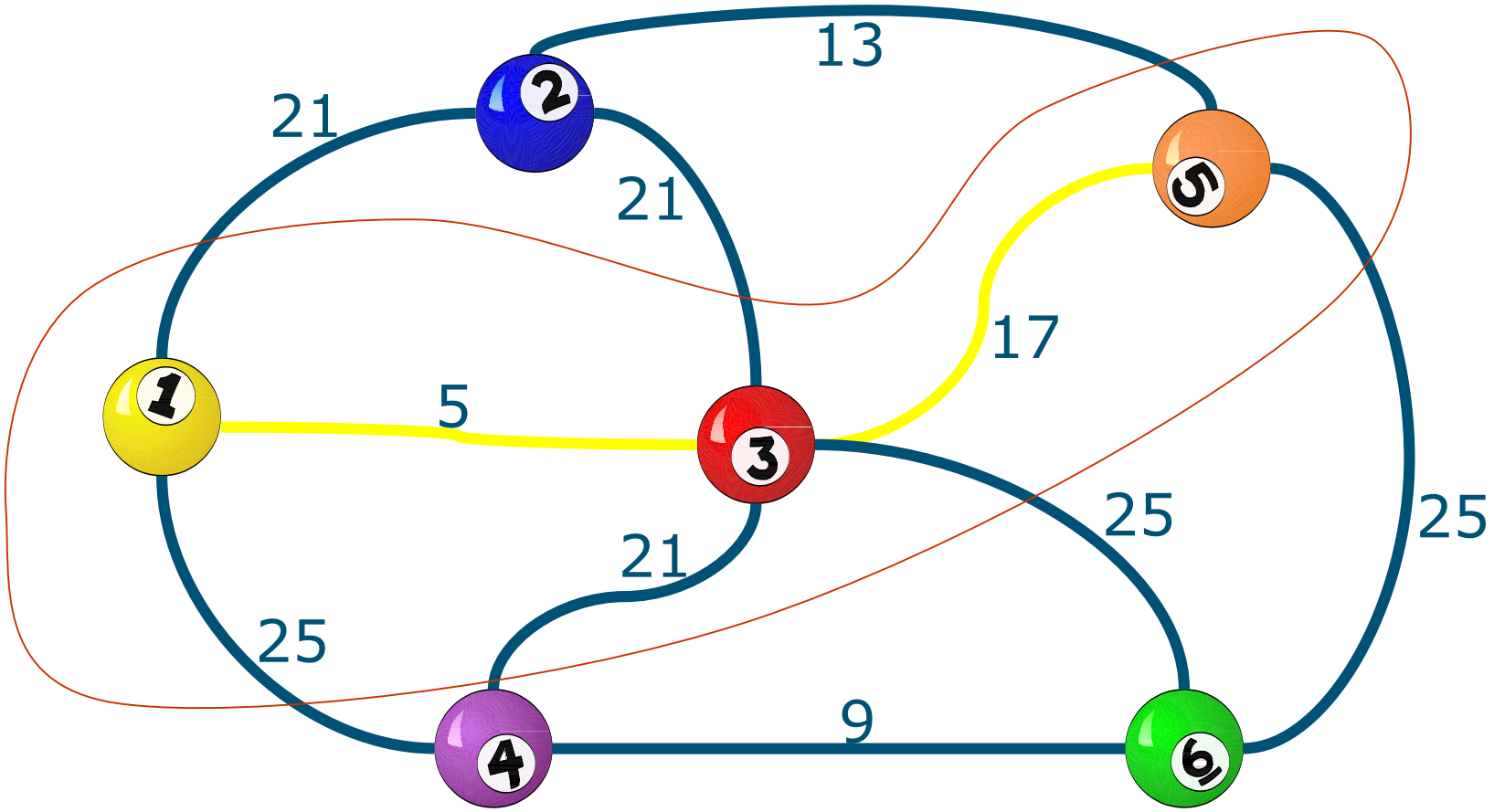
Algoritmos Greedy en grafos. ARCM (**Prim**)



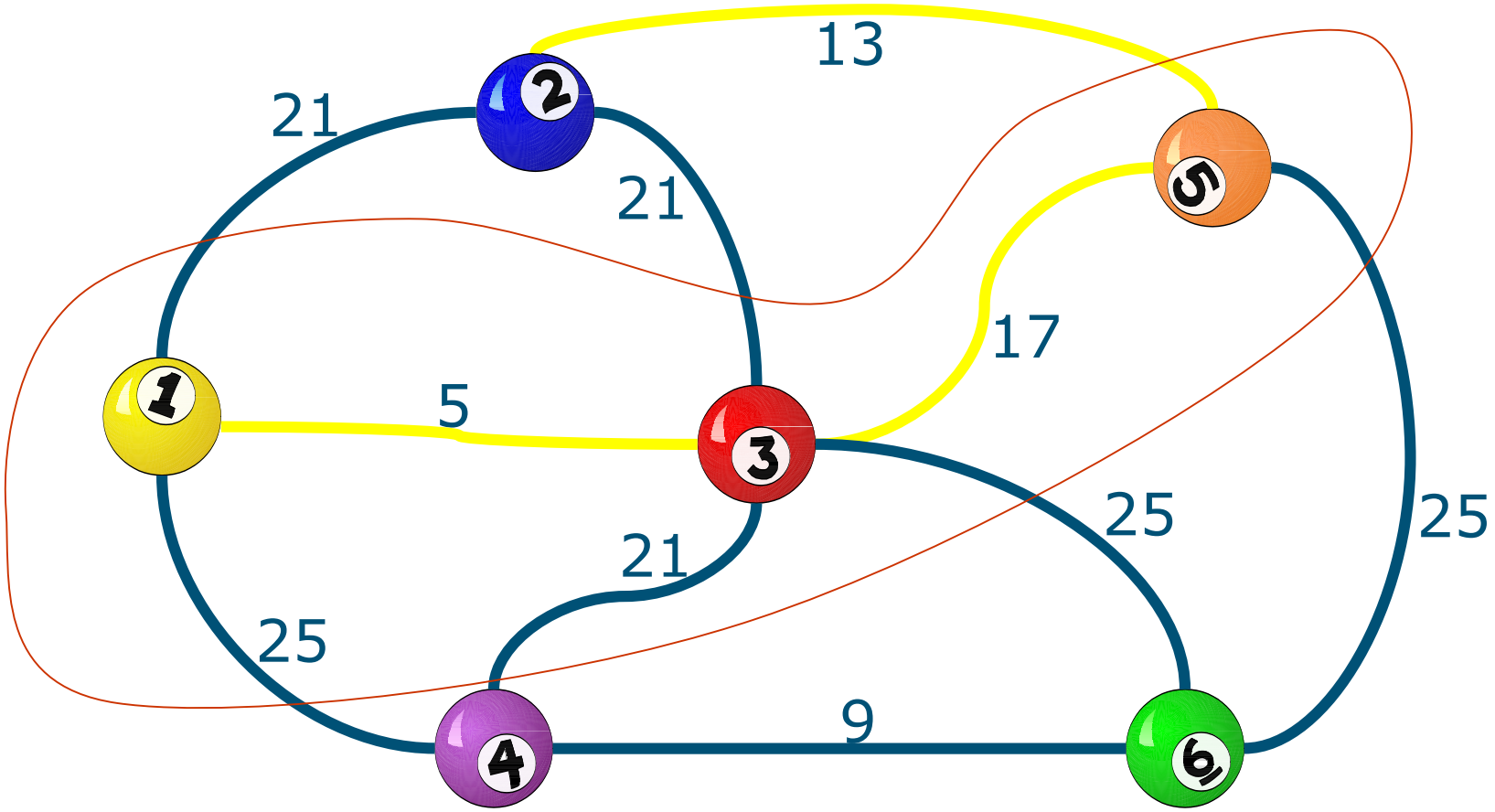
Algoritmos Greedy en grafos. ARCM (**Prim**)



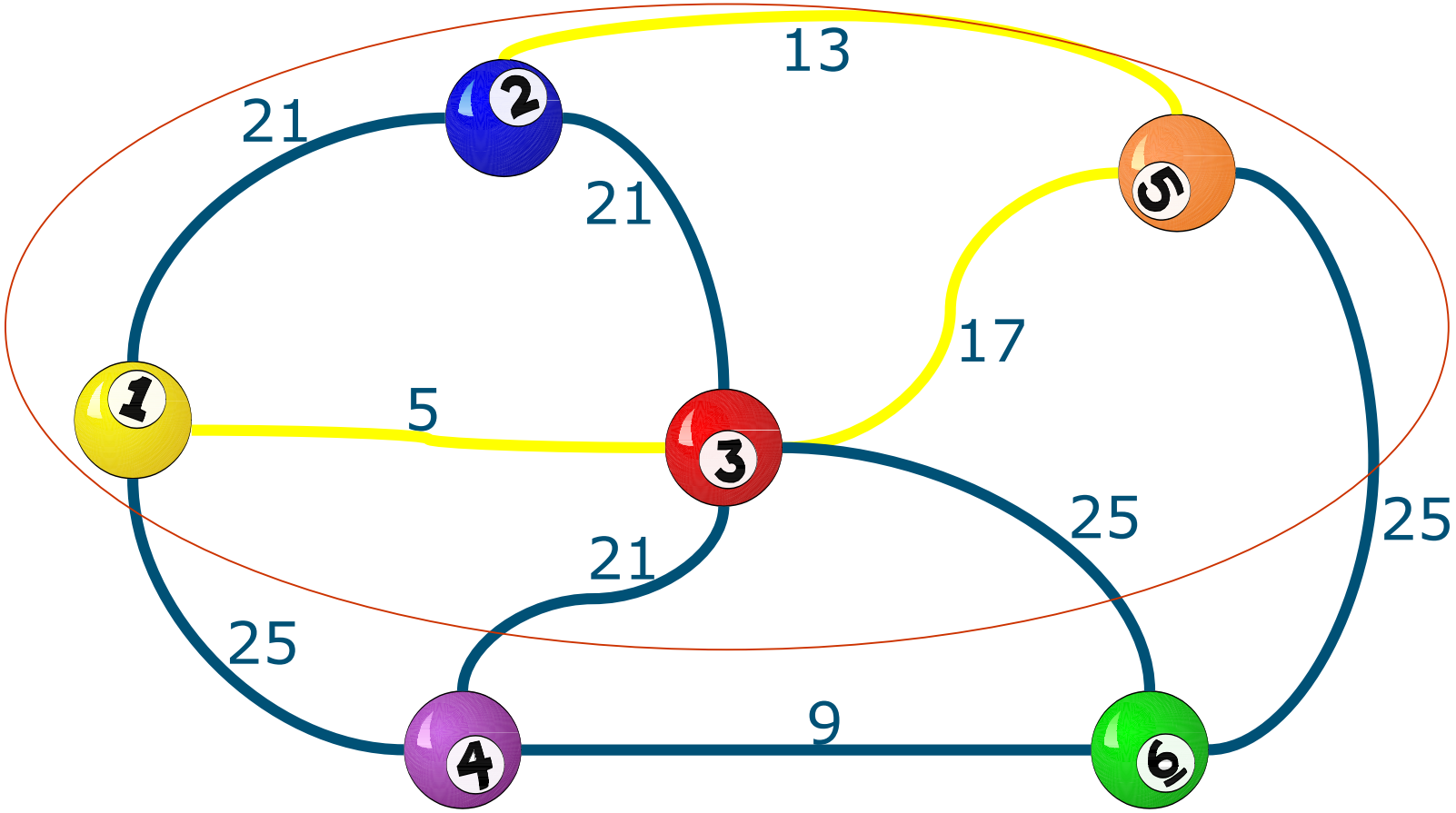
Algoritmos Greedy en grafos. ARCM (**Prim**)



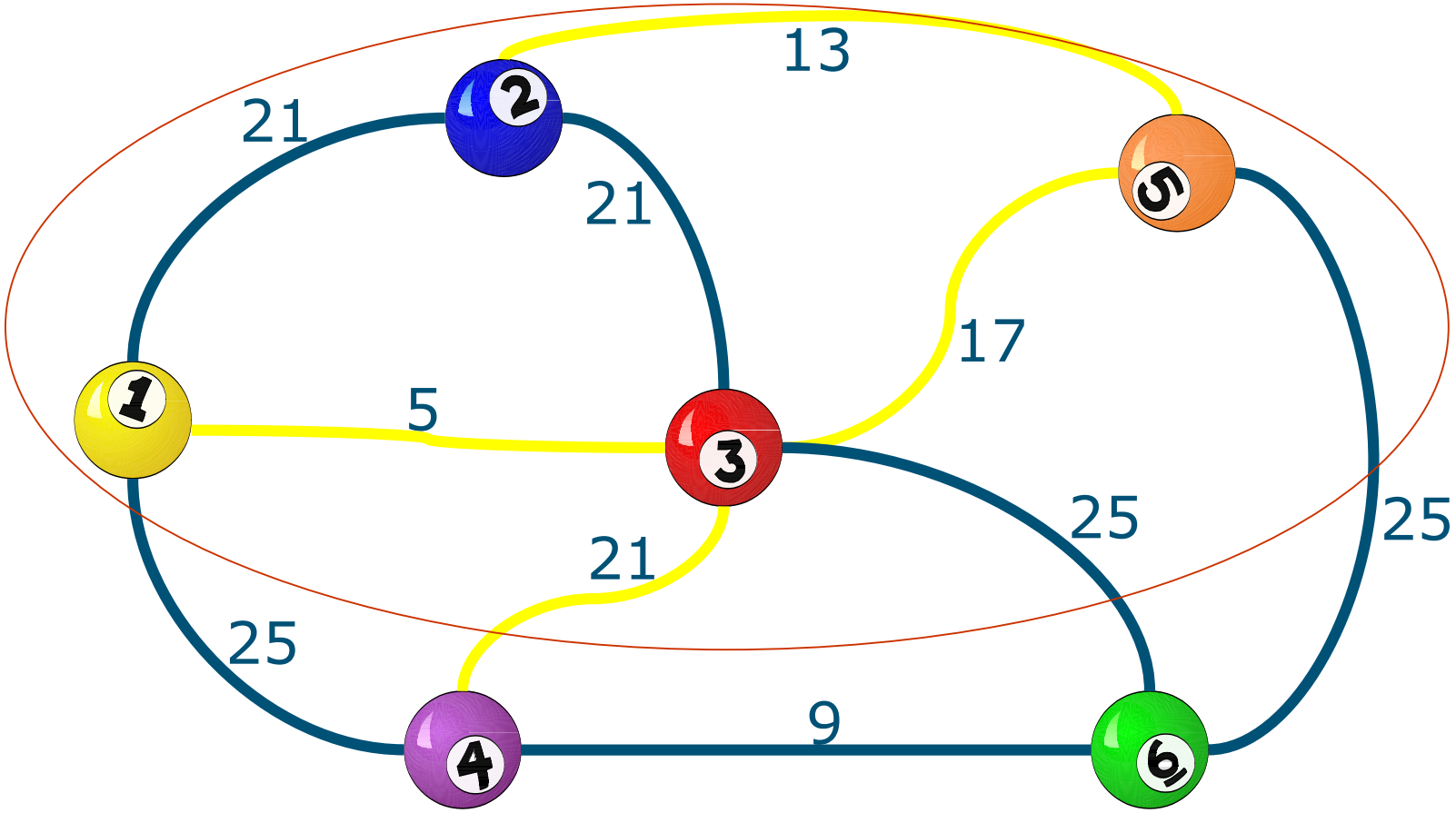
Algoritmos Greedy en grafos. ARCM (**Prim**)



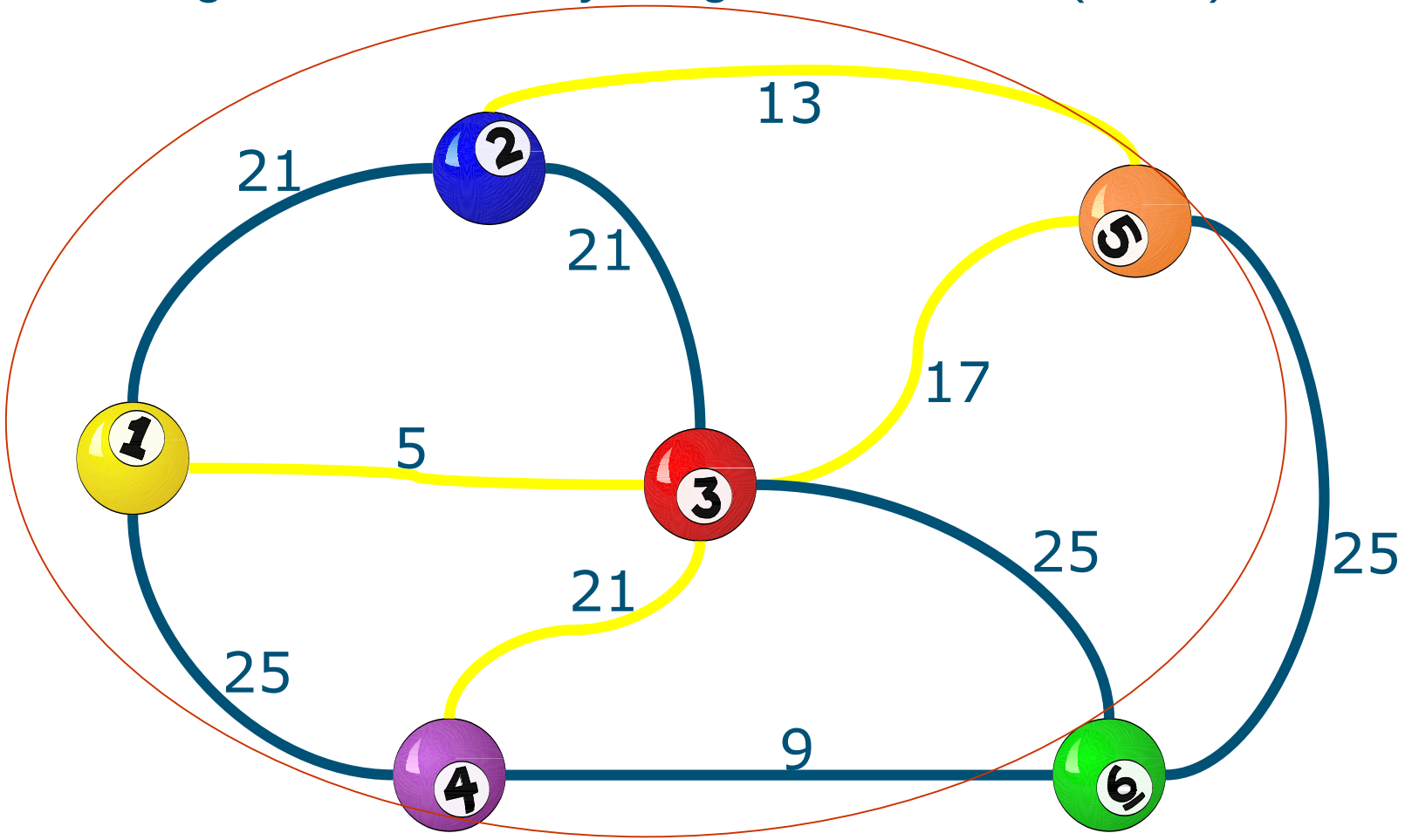
Algoritmos Greedy en grafos. ARCM (**Prim**)



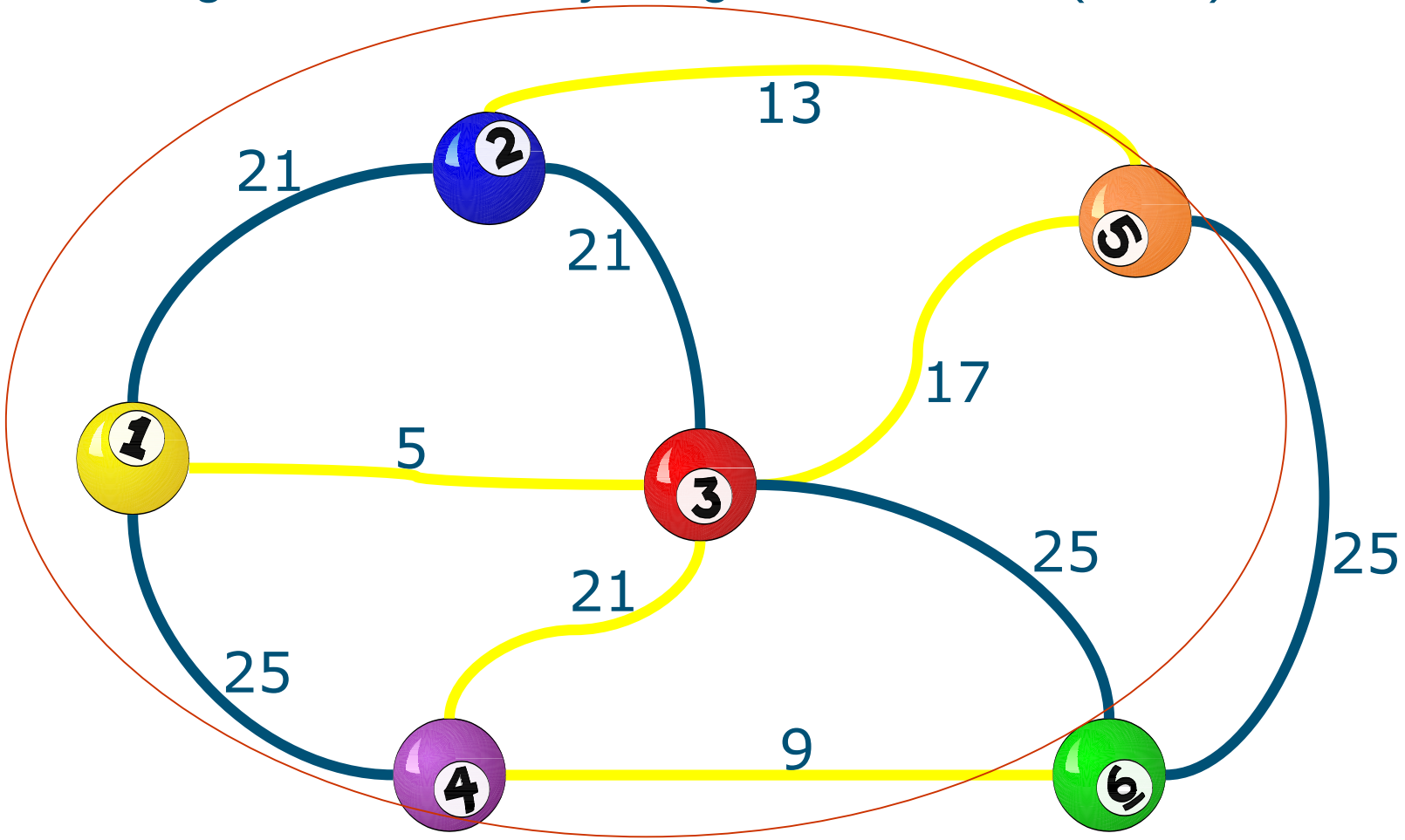
Algoritmos Greedy en grafos. ARCM (**Prim**)



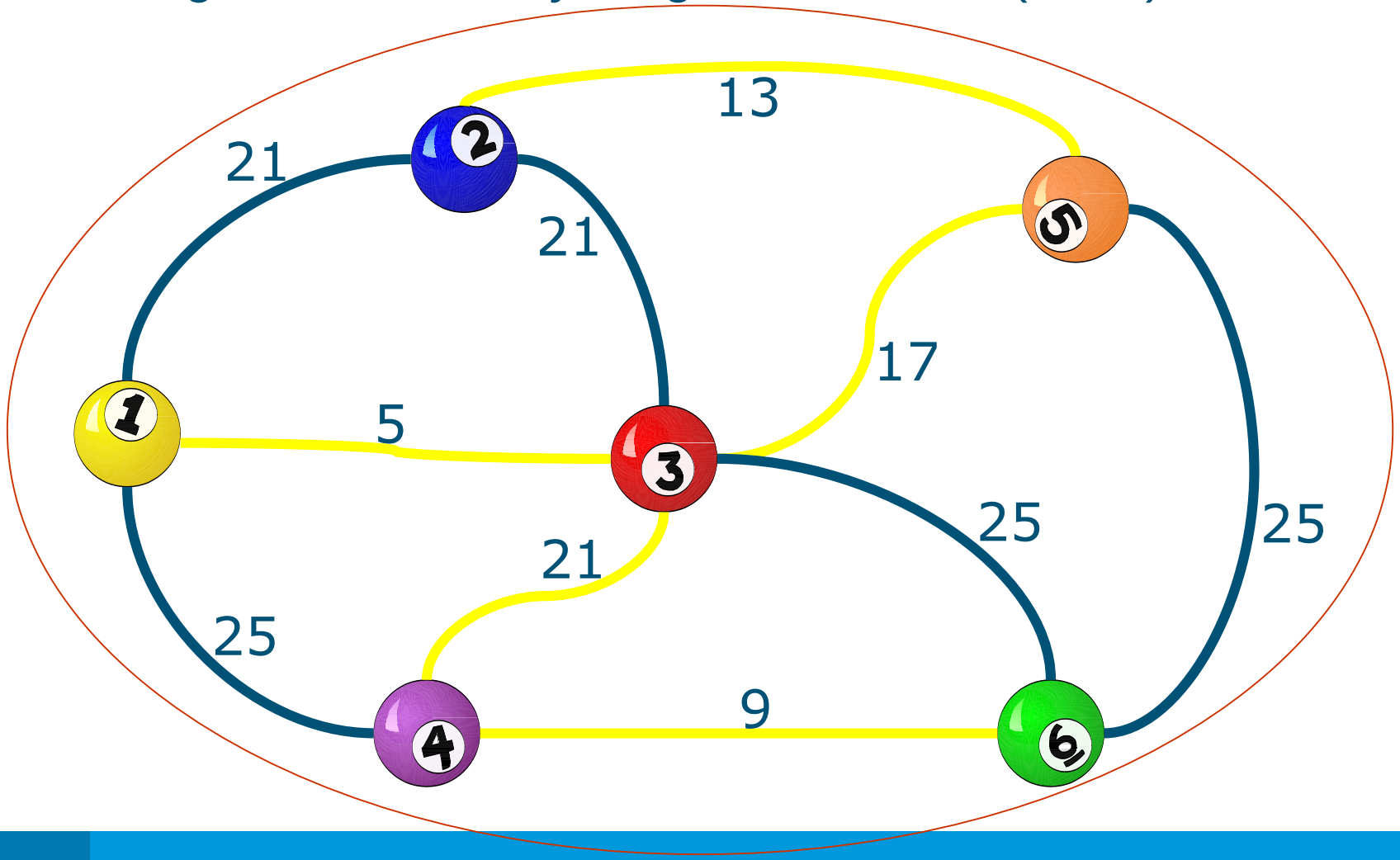
Algoritmos Greedy en grafos. ARCM (**Prim**)



Algoritmos Greedy en grafos. ARCM (**Prim**)



Algoritmos Greedy en grafos. ARCM (**Prim**)



Algoritmos Greedy en grafos. ARCM (**Kruskal**)

- Dado un grafo G = (V, E), se empieza con un grafo T = {V, ∅} ⇒ T ≡ grafo con todos los vértices de G, pero sin ninguna arista
- Durante el algoritmo se mantendrá siempre un **bosque de árboles**. Inicialmente, el bosque contiene tantos árboles como vértices tiene el grafo ⇒ cada uno de los árboles está formado por un único vértice
- Para añadir las aristas se examina el conjunto E en orden creciente según su peso ⇒ Ordenar las aristas del grafo según su peso y una buena estructura de datos para ello es una cola de prioridad

Algoritmos Greedy en grafos. ARCM (**Kruskal**)

- Proceso de selección de arista y construcción del árbol de recubrimiento de coste mínimo:
  - Si la arista seleccionada conecta dos vértices de árboles distintos, se añade al conjunto de aristas de T puesto que formará parte del árbol de recubrimiento y ambas componentes se unen en una única componente conexa
  - Si la arista conecta dos vértices que pertenecen al mismo árbol, no se añade para evitar la aparición de ciclos en el árbol de recubrimiento
- Cuando todos los vértices de T estén en una única componente conexa, se habrá encontrado el árbol de recubrimiento de coste mínimo buscado

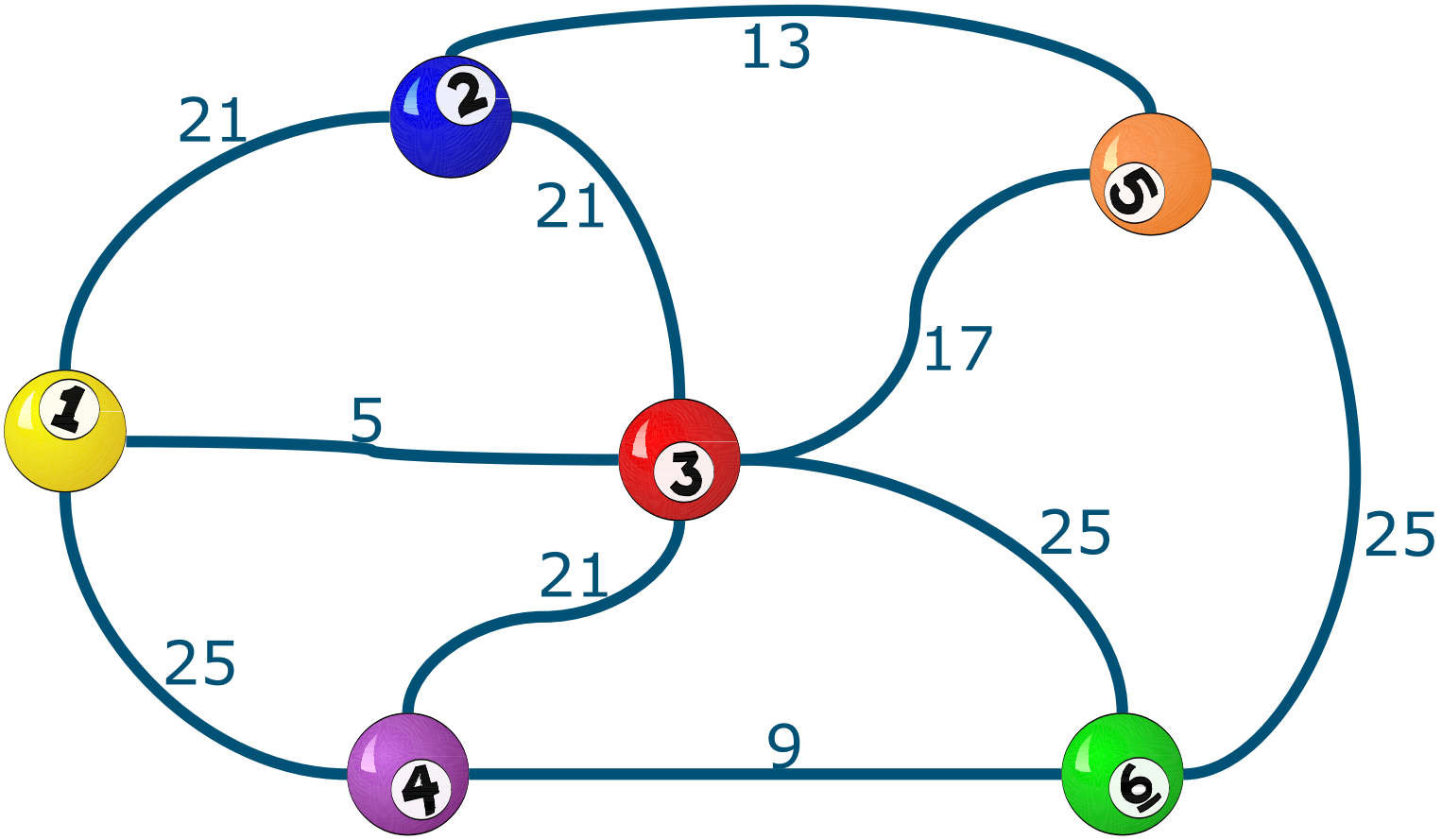
Algoritmos Greedy en grafos. ARCM (**Kruskal**)

```
funcion Kruskal(Grafo G)
    Ordenar las aristas E de G por pesos crecientes
    Iniciar n conjuntos disjuntos // uno para cada vértice de V
    n = número de vértices de V
    MST = ∅
    repetir  // bucle voraz
        e = arista(u, v) de menor peso no considerada aún
        conj_u = conjunto disjunto que contiene al vértice u
        conj_v = conjunto disjunto que contiene al vértice v
        si (conj_u ≠ conj_v)
            fusionar(conj_u, conj_v)
            MST = MST ∪ {e}
    hasta que MST contenga n − 1 aristas
    devolver MST
```

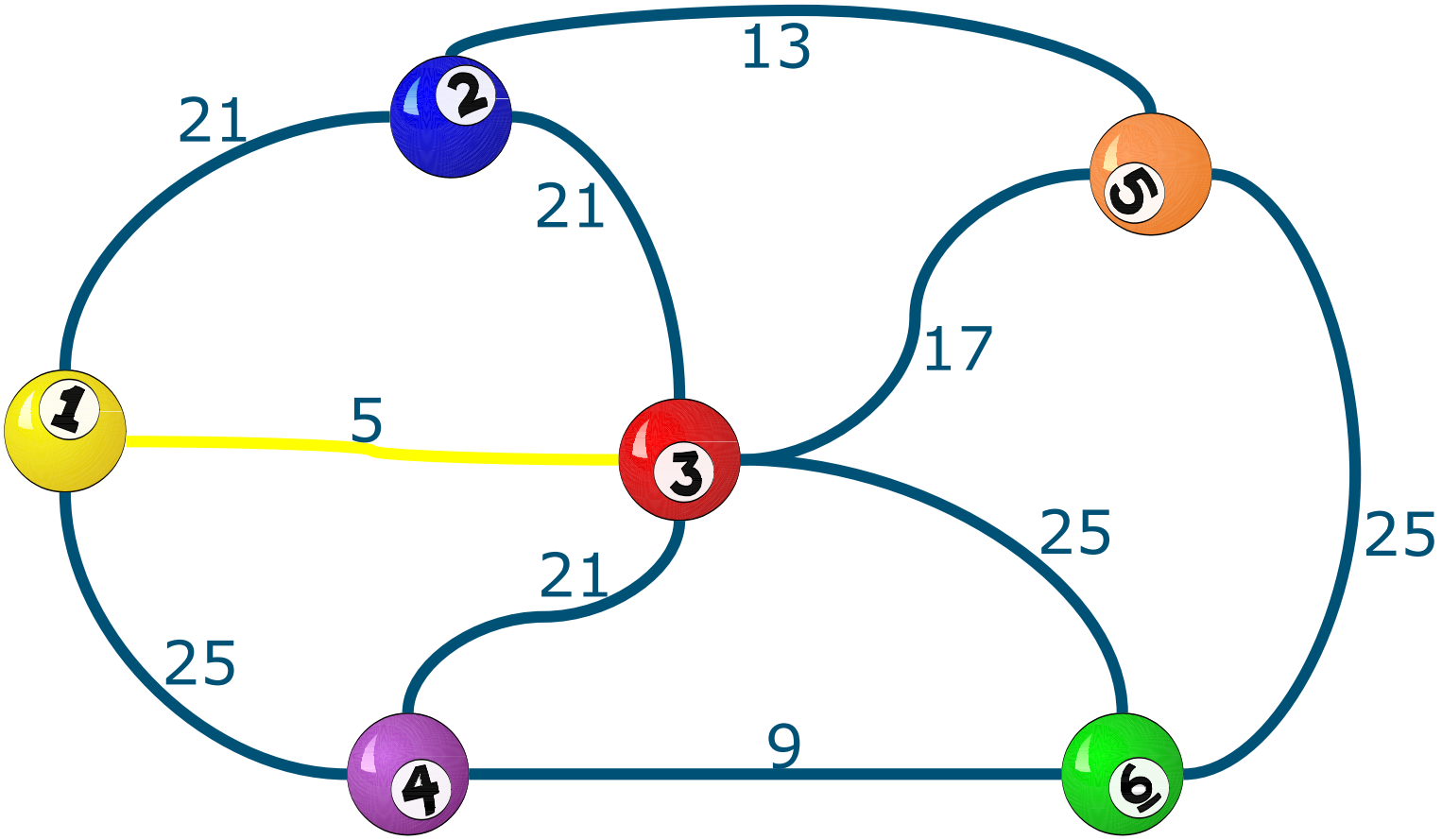
Algoritmos Greedy en grafos. ARCM (**Kruskal**)

- Tiempo de ejecución:
  - **Ordenar las aristas**: O(m log m), equivale a O(m log n), ya que n − 1 ≤ m ≤ n*(n − 1)/2 (n es el número de vértices y m el de aristas)
  - Inicializar los n conjuntos disjuntos: O(n)
  - Bucle principal **repetir** se ejecuta n - 1 veces
    - Operaciones obtener y fusionar: en el caso peor
      - 2m en las operaciones de obtener los conjuntos y (n - 1) para fusionarlos, en conjuntos disjuntos estas operaciones se ejecutan en O(log n)
  - $T_{Kruskal}(n, m) \in O(m \log n)$
- Mejora: usar un heap (montículo) mínimo para ordenar las aristas por su peso (la de menos peso estaría en la raíz)

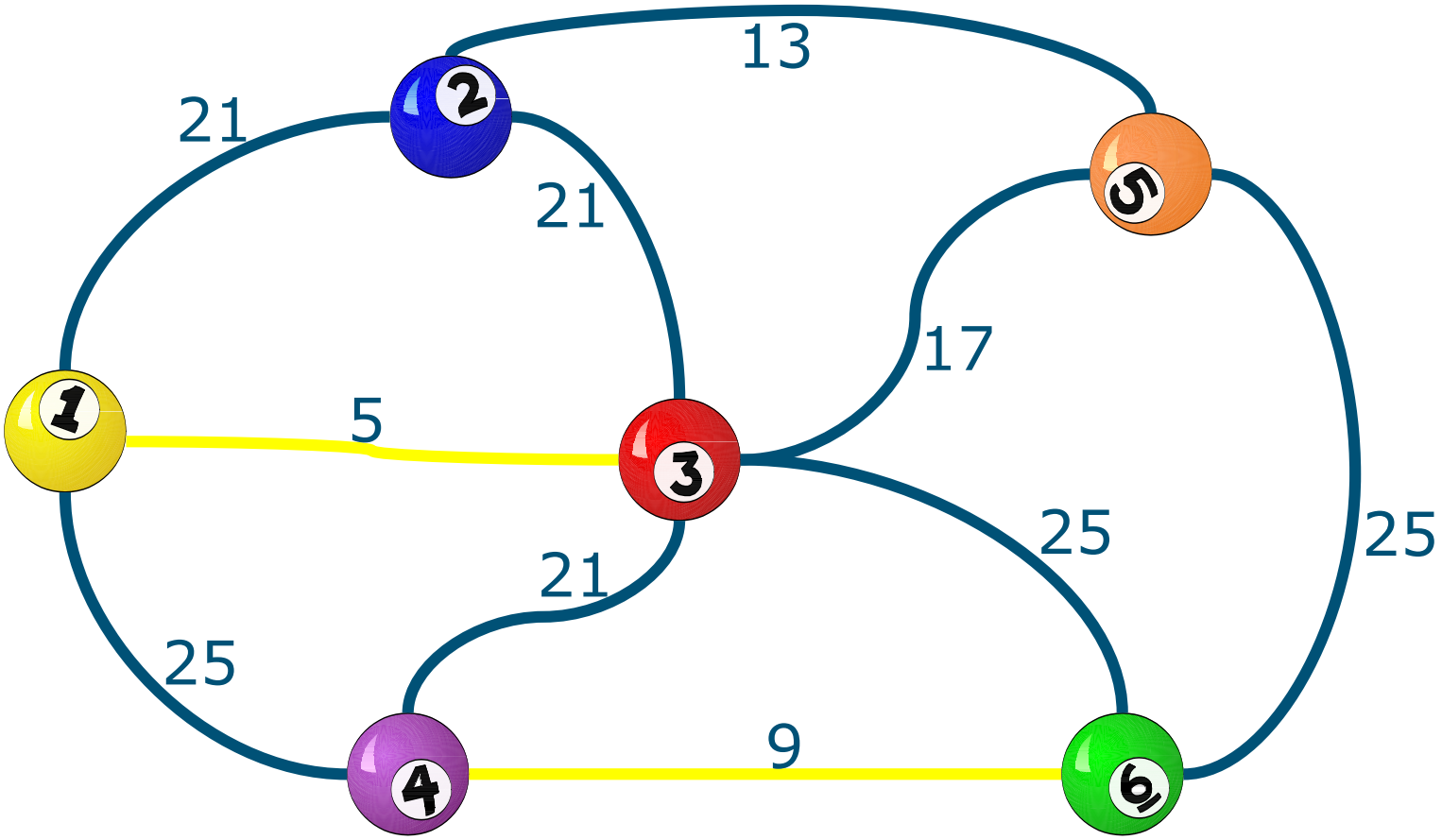
Algoritmos Greedy en grafos. ARCM (**Kruskal**)



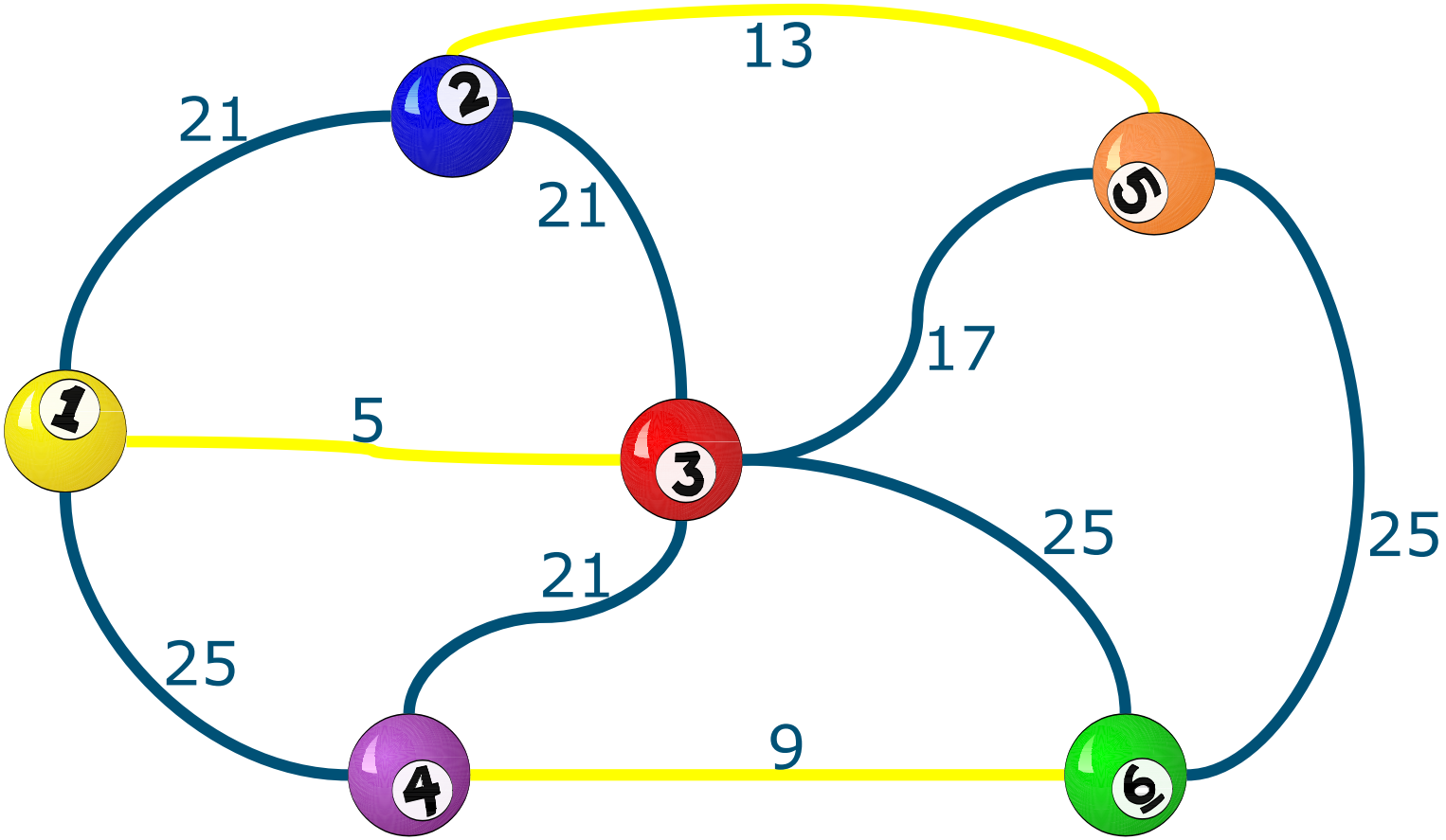
Algoritmos Greedy en grafos. ARCM (**Kruskal**)



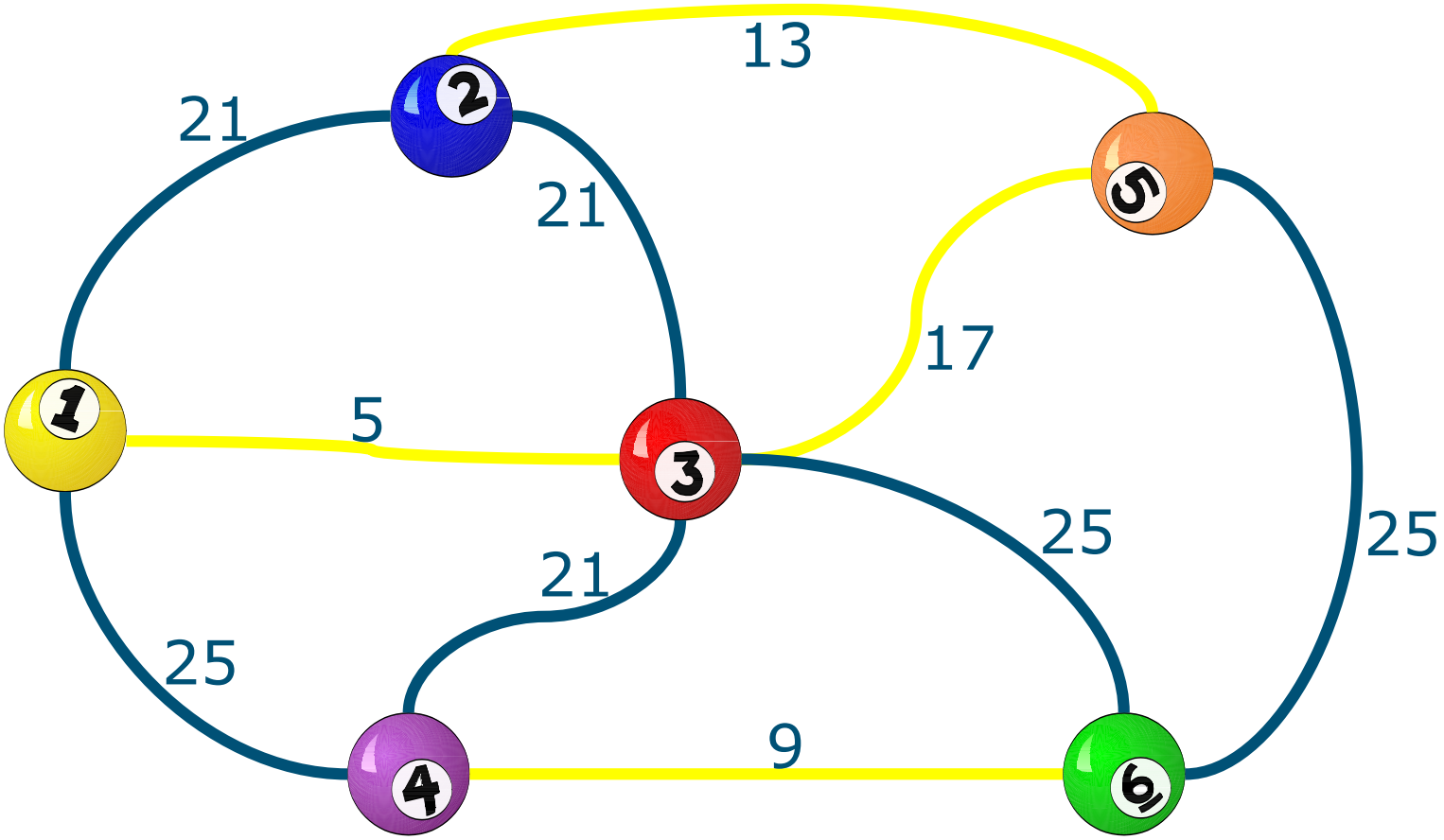
Algoritmos Greedy en grafos. ARCM (**Kruskal**)



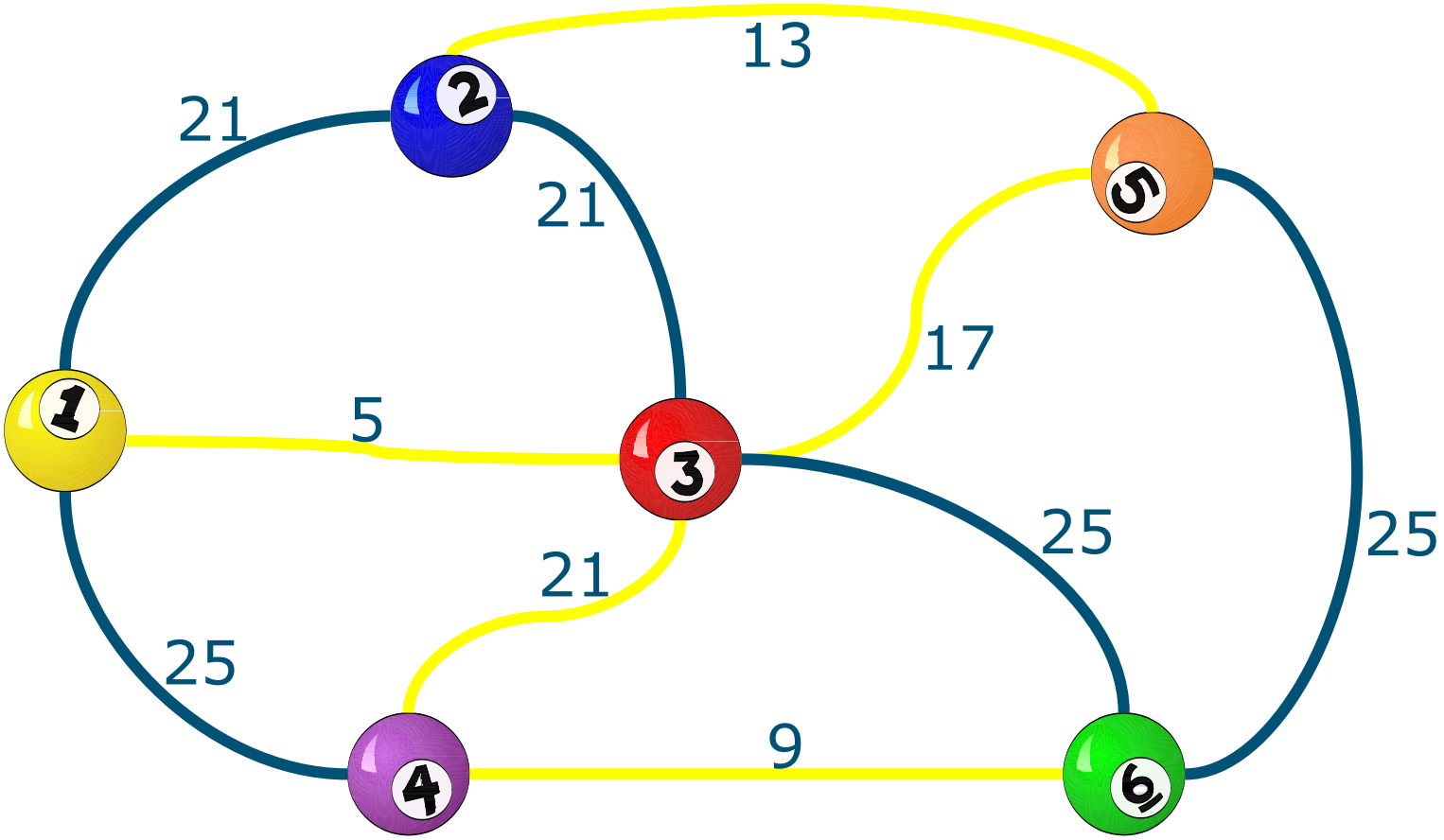
Algoritmos Greedy en grafos. ARCM (**Kruskal**)



Algoritmos Greedy en grafos. ARCM (**Kruskal**)



Algoritmos Greedy en grafos. ARCM (**Kruskal**)



Algoritmos Greedy en grafos. Resumen

- Siendo: n = número de vértices de G, m = número de aristas
- **Dijkstra** $\Rightarrow$ O((n+m) log n) com heap (PQ), peor caso $O(n^2)$
- **Prim** $\Rightarrow$ O(m log n) con heap (PQ), peor caso $O(n^2)$
- **Kruskal** $\Rightarrow$ O(m log n) con heap (PQ), peor caso $O(n^2)$
- Para el ARCM qué algoritmo utilizar ¿Prim o Kruskal?
  - Sabemos que $n - 1 \leq m \leq (n*(n - 1)) / 2$
  - Si el **grafo es muy denso** ($m \rightarrow n(n-1)/2$), entonces el algoritmo de Kruskal requiere $O(n^2 \log n)$, por lo que el algoritmo de Prim puede ser mejor
  - Si el **grafo es disperso** ($m \rightarrow n$), entonces el algoritmo de Kruskal requiere O(n log n), por lo que el algoritmo de Prim puede ser menos eficiente

Problema de la mochila (Knapsack Problem)

- Tenemos una **mochila** capaz de albergar hasta un peso máximo W. Existen n de objetos o_i ($0 \leq i \leq n$) que podemos cargar en la mochila, cada uno con un peso w_i y un valor de beneficio asociado v_i. Queremos encontrar las proporciones de los n objetos $x_1, x_2, ..., x_n$ ($0 \leq x_i \leq 1$) que tenemos que introducir en la mochila de forma que la suma de los beneficios de los objetos escogidos sea máxima. Es decir, encontrar valores $(x_1, x_2, ..., x_n)$ de forma que se maximice la cantidad $\sum_{i=1}^{n} v_i * x_i$, sujeta a la restricción $\sum_{i=1}^{n} w_i * x_i \leq W$
- Los objetos no se pueden fraccionar ⇒ **mochila 0/1**
- Los objetos se pueden fraccionar, transportando solo parte ⇒ **mochila 0-1**

Problema de la mochila 0/1

- Los objetos no se pueden fraccionar ⇒ **mochila 0/1**
- Problema complejo (NP). Solución greedy heurística, no siempre consigue la solución óptima

- $x_i = 0$ o $x_i = 1$ ($1 \leq i \leq n$), siendo n el número de objetos
- v_i ($v_i > 0$) es el valor (beneficio) asociado al objeto i
- w_i ($w_i > 0$) es el peso del objeto i
- W es el peso máximo soportado por la mochila
- **Objetivo**: encontrar valores (x_1, x_2, ..., x_n) de forma que se maximice la cantidad $\sum_{i=1}^{n} v_i * x_i$, sujeta a la restricción $\sum_{i=1}^{n} w_i * x_i \leq W$

Problema de la mochila 0/1

- **Mochila 0/1**
- Heurística greedy
- Ordenar los objetos por densidad (razón **beneficio/peso**) decreciente

$$\frac{v_i}{w_i} \geq \frac{v_{i+1}}{w_{i+1}} \text{ para } 1 \leq i \leq n$$

- Si C es la cantidad máxima de los objetos que se pueden llevar en la mochila, la heurística garantiza obtener, al menos, un valor de C/2
- La heurística puede proporcionar resultados mucho peores (muy alejados del óptimo) para instancias particulares del problema

Problema de la mochila 0/1

```
funcion Mochila1(double [1..n] w, double [1..n] v, double W): boolean [1..n]x
    para i = 1 hasta n
        x[i] = false
    peso = 0
    mientras  (peso < W)
        i = mejorObjetoRestante()
        si (peso + w[i] < W)
            x[i] = true
            peso = peso + w[i]
    devolver x
```

Eficiencia del algoritmo:

- Sin ordenar los objetos en base a la relación v/w ⇒ Bucle externo (n iteraciones) requiere O(n); y buscar el objeto con mejor relación v/w requiere O(n) ⇒ $O(n^2)$
- Ordenando los elementos en base a la relación v/w ⇒ Ordenar los objetos en base a la relación v/w requiere O(n log n); bucle externo requiere O(n); y ahora la función de selección requiere O(1) ⇒ O(n log n)

Problema de la mochila 0-1

- Los objetos se pueden fraccionar ⇒ **mochila 0-1**

Esquema general Greedy

- Candidatos **C**: diferentes objetos i tal que 1 ≤ i ≤ n
- Conjunto **solución**: vector (x_1, x_2, ..., x_n), indica la fracción que se toma de cada objeto
- Conjunto es **factible**: si cumple que v_i, w_i > 0, 0 ≤ x_i ≤ 1 y la restricción $\sum_{i=1}^{n} w_i * x_i \leq W$
- Función **objetivo**: maximizar el valor total de los objetos de la mochila
- Función de **selección**:
  - Caso ideal: meter todos los objetos en la mochila

Problema de la mochila 0-1

- Función de **selección**:
  - Solución óptima: llenar exactamente la mochila $\sum_{i=1}^{n} w_i = W$
  - General: Ordenar los objetos y seleccionarlos de forma que se incluya la mayor fracción posible de cada uno. Finaliza cuando se alcance W
- Recordar que:
  - Si se pone una fracción x_i del objeto i en la mochila: se consigue un beneficio $v_i * x_i$, $0 \leq i \leq 1$ y aporta un peso de $w_i * x_i$
  - El objetivo es: maximizar $\sum_{i=1}^{n} v_i * x_i$ con la restricción $\sum_{i=1}^{n} w_i * x_i \leq W$

Problema de la mochila 0-1

```
funcion Mochila2(double [1..n] w, double [1..n] v, double W): boolean [1..n]x
    para i = 1 hasta n
        x[i] = 0
    peso = 0
    mientras  (peso < W)
        i = mejorObjetoRestante()
        si (peso + w[i] ≤ W)
            x[i] = 1
            peso = peso + w[i]
        si no
            x[i] = (W - peso) / w[i]
            peso = W
    devolver x
```

Función **mejorObjetoRestante()**

- Obtener el objeto más valioso, mayor v_i (aumento el valor de la carga más rápidamente)
- Obtener el objeto más ligero, menor w_i (aumento del peso total lo más lento posible)
- Objeto cuyo valor por unidad de peso, v_i / w_i, sea el mayor posible

Problema del viajante (Travelling-salesman Problem)

- **Objetivo** (problema del viajante): Dado un grafo G = (V, E), totalmente conexo (completo), no dirigido y ponderado, encontrar un camino que empiece en un vértice v ∈ V y acabe en ese mismo vértice, pasando una única vez por todos los vértices de V (es decir, un circuito hamiltoniano), con el objetivo de que sea el circuito hamiltoniano de coste mínimo
- Se trata de encontrar el **itinerario más corto** que permita a un viajante recorrer un conjunto de ciudades, pasando una única vez por cada una y regresando a la ciudad de partida. Se suponen conocidas las distancias entre cada par de ciudades
- Éste es un problema **NP-completo** ⇒ Encontrar solución exacta orden exponencial (explosión combinatoria) ⇒ Resolverlos con algoritmos heurísticos ⟶ **heurística greedy**

Problema del viajante

- **Algoritmo fuerza bruta** ⇒ Calcular el coste de todas las posibles rutas y elegir la de longitud mínima: crece exponencialmente con el número de vértices
- **Heurística Greedy 1 ⇒ Los vértices son los candidatos** ⇒ Se empieza en un nodo cualquiera y en cada paso nos vamos al nodo no visitado más próximo al último nodo seleccionado. Se evitará cerrar ciclos salvo en el último momento
- **Heurística Greedy 2 ⇒ Las aristas son los candidatos** ⇒ Se incluirá en cada paso la arista de menor peso, pero garantizando que se pasa dos veces por el mismo vértice y que con la última arista se cierra el ciclo → Hacer como en el algoritmo de *Kruskal*, pero garantizando que se forme un ciclo

Problema del viajante

- **Heurística Greedy 1** ⇒ Candidatos = **V**
- Una **solución** será un cierto orden en el conjunto de vértices
- **Representación de la solución** (orden en el conjunto de vértices): $S = (v_1, v_2, ..., v_n)$, donde v_i es el vértice visitado en el lugar i-ésimo
- **Inicialización**: empezar en un vértice cualquiera
- **Función de selección**: de los vértices candidatos seleccionar el más próximo al último (o al primero) de la secuencia actual $(v_1, v_2, ..., v_{actual})$
- **Objetivo**: minimizar el coste del circuito que une los n vértices
- Acabamos cuando tengamos n vértices

Problema del viajante

- **Heurística Greedy 2** ⇒ Candidatas las **aristas**
- Seleccionar la pareja de vértices con menor distancia
- A continuación, se selecciona la siguiente pareja de vértices con distancia mínima tal que:
  - no se visite un vértice dos o más veces
  - no se cierre el recorrido antes de haber visitado todos los nodos
- De esta forma, si hay que visitar n vértices, se seleccionan n parejas de vértices que serán visitados de forma consecutiva, y la solución consiste en **reordenar** esas parejas de forma que **constituyan un recorrido**

Problema del viajante

- **Heurística Greedy 2** ⇒ Candidatas las **aristas**
- Se ordenan las aristas y se procede como en el algoritmo de Kruskal, pero garantizando que al final se forme un circuito. En este caso, la eficiencia depende del algoritmo de ordenación
- En cada etapa, se elige la arista más corta de las restantes, teniendo en cuenta que:
  - No forme un ciclo con las aristas ya escogidas (excepto para la última arista elegida, que completará el recorrido del viajante)
  - La elegida no sea la tercera arista incidente a un nodo cualquiera (formando así una estrella)
- **Solución**: conjunto de aristas $s = (a_1, a_2, ..., a_n)$ que formen un ciclo hamiltoniano, sin importar el orden, $a_i =$

Problema del viajante

- **Heurística Greedy 2** ⇒ Candidatos = **E**
- Una **solución** será un conjunto de aristas (a_1, a_2, ..., a_n) que formen un ciclo hamiltoniano, sin importar el orden
- **Representación de la solución**: S = (a_1, a_2, ..., a_n), donde cada a_i es una arista, de la forma a_i = (v_i, w_i)
- **Inicialización**: empezar con un grafo sin aristas
- **Función de selección**: arista candidata de menor coste
- **Función de factibilidad**: una arista se puede añadir a la solución actual si no se forma un ciclo (excepto para la última añadida) y si los nodos unidos no tienen grado mayor que dos
- **Objetivo**: minimizar coste del circuito que forman las n aristas

Problema del viajante

- **Conclusiones**
- Ninguno de las dos aproximaciones algorítmicas garantiza la solución óptima
- No obstante, *normalmente* ambos dan soluciones buenas, próximas a la óptima
- Posibles mejoras:
  - Buscar heurísticas mejores, más complejas
  - Repetir la heurística 1 con varios orígenes
  - A partir de la solución del algoritmo intentar hacer modificaciones locales para mejorar esa solución

Planificación de Tareas

- **Planificación de tareas con fecha límite** (scheduling with deadlines)
- **Objetivo** ⇒ Seleccionar un conjunto de tareas factibles con fecha límite de ejecución maximizando el beneficio
- **Planteamiento**: Tenemos un procesador y n posibles tareas para ejecutar
- Cada tarea i ($1 \leq i \leq n$) tiene una **fecha límite** d_i (o deadline), es decir, la tarea i tiene que estar finalizada antes, o en el propio instante de tiempo d_i, o sea, en un instante de tiempo $t \leq d_i$
- Todas las tareas consumen la **misma cantidad de tiempo**: una unidad o instante de tiempo (time slot)

Planificación de Tareas

- En cada **instante** (time slot), solo se puede ejecutar una tarea, bajo la suposición de que en ese instante se realiza la tarea completa (el procesador no es compartido)
- Suponemos también que la tarea i produce una ganancia o beneficio b_i ($b_i > 0$) si y solo si se ejecuta sin sobrepasar su fecha límite (deadline), es decir, sin sobrepasar el instante de tiempo d_i, en un instante $t \leq d_i$
- Puede que no sea posible ejecutar todas las tareas
- **Objetivo**: establecer una secuencia posible de tareas (s_1, s_2, ..., s_m), respetando las fechas límite (deadlines), que permita maximizar el beneficio obtenido (suma de los beneficios asociados a las tareas de la secuencia)

$$b_{Total} = \sum_{i=1}^{m} b_{s_i}$$

Planificación de Tareas

- **Algoritmo sencillo**: Comprobar todos los posibles órdenes de las tareas, calcular los beneficios asociados de las ordenaciones posibles y quedarse con la de mayor beneficio (que sea factible) ⇒ habría que generar las **n!** posibles ordenaciones de tareas y ver las que se son factibles ⇒ El algoritmo tendría una complejidad de O(n!)
- Una **solución** estará formada por un conjunto de candidatos, junto con un orden de ejecución de los mismos
- Un conjunto de tareas es **factible** si existe al menos una **secuencia** (también factible) que permite que todas las tareas en el conjunto se ejecuten en el tiempo de sus respectivas fechas límite

Planificación de Tareas

- **Ejemplo**: Sean las cuatro tareas **n** = 4, con beneficios **b** = (100, 10, 15, 27) y con tiempos límites **d** = (2, 1, 2, 1). La tarea 1 con deadline 2, indica que la tarea 1 puede empezar en el instante 1 o en el 2. Mientras que la tarea 2 tiene deadline 1, dicha tarea puede empezar solo en el instante 1
- **Posibles soluciones**: (1, 2) (1, 3) (1, 4) (2, 1) (2, 3) (2, 4) …

T	1	2
S	1	3
b	100	15
d	2	2

$b_{Total} = 115$

T	1	2
S	4	3
b	27	15
d	1	2

$b_{Total} = 42$

T	1	2
S	1	4
b	10	27
d	2	1

No factible

T	1	2
S	4	1
b	27	100
d	1	2

$b_{Total} = 127$

Planificación de Tareas

- **Aclaración**: Ordenaciones no factibles serían: (1, 2) (1, 4) (2, 4) (3, 2) (3, 4) (4, 2). La ordenación (1, 2) no es posible porque la tarea 1 empezaría primero en el instante 1, tomaría un slot de tiempo para terminar (deadline = 2), provocando que la tarea 2 empiece en el instante 2. Sin embargo, el deadline de la tarea 2 es el instante 1. Por lo que las ordenaciones que tengan en segundo lugar la tarea 2 no serán factibles. Igual sucede con la tarea 4. Las ordenaciones posibles son: (1, 3) [115] (2, 1) [110] (2, 3) [25] (3, 1) [115] **(4, 1) [127]** (4, 3) [42]. La ordenación (1, 3) es posible porque la tarea 1 se empieza antes de su deadline, y la tarea 3 se empieza en su deadline (2). La ordenación (4, 1) es la óptima con 127 de beneficio

Planificación de Tareas

- **Aplicamos un algoritmo greedy** ⇒ En el que empezamos con una planificación sin tareas y vamos añadiéndolas paso a paso la tarea de mayor beneficio b_i entre aquellas que aún no se han considerado
- Una **solución** estará formada por un conjunto de candidatos, junto con un orden de ejecución de los mismos
- **Representación de la solución**: $S = (s_1, s_2, ..., s_m)$, donde s_i es la tarea ejecutada en el instante i
- **Función de selección**: de los candidatos restantes a elegir, se selecciona el que tenga mayor valor de beneficio: **argmax{b_i}**
- ¿Cómo es la función **factible (S, x)**?

Planificación de Tareas

- **Planificación actual**

T	1	2	3	4
S	s_1	s_2	s_3	s_4
b	b_{s_1}	b_{s_2}	b_{s_3}	b_{s_4}
d	d_{s_1}	d_{s_2}	d_{s_3}	d_{s_4}

x
b_x
d_x

- ¿Dónde debería ser colocada **x** dentro de la planificación?
- ¿Es factible la solución parcial que incluye a **x**?
- **Idea 1**: Probar todas las posibles colocaciones ⇒ NO
- **Idea 2**: Ordenar las tareas por orden de deadline $\mathbf{d_x}$ ⇒ Que las tareas con deadlines más tempranos se ejecuten antes ⇒ OK

Planificación de Tareas

- **Lema** ⇒ Sea **J** un conjunto de **k** tareas. Existe una ordenación factible de **J** (es decir que respeta los deadlines) si y sólo si la ordenación $S = (s_1, s_2, ..., s_k)$, co

T	1		2		...		k
S	s_1		s_2		...		s_k
b	b_{s1}		b_{s2}		...		b_{sk}
d	d_{s1}	≤	d_{s2}	≤	...	≤	d_{sk}

- Es decir, sólo es necesario probar la planificación en orden creciente de deadline de ejecución, comprobando que cada $\mathbf{d_{si} \geq i}$ (la tarea ejecutada en la i-ésima posición tiene un deadline de **i** ó más)

Planificación de tareas

- **Teorema** ⇒ El método greedy consiste en incluir en cada la tarea con mayor b_i de modo que la solución parcial obtenida sea realizable lleva a una solución óptima
- **Restricción** ⇒ Cada vez que se selecciona una nueva tarea para incluirla en la solución parcial hay que comprobar si existe alguna ordenación que respete los deadlines de ejecución
- **Estructura del algoritmo greedy**
- **Inicialización**: Empezar con una secuencia vacía, con todas las tareas como candidatas
- **Ordenar** las tareas según el valor de b_i

Planificación de tareas

- **En cada paso, hasta que se acaben los candidatos, repetir:**
- **Selección**: Elegir entre los candidatos restantes el que tenga mayor beneficio
- **Factible**: Introducir la nueva tarea en la posición adecuada, según los valores de deadline **d**
  - Si el nuevo orden (s_1, s_2, ..., s_k) es tal que $\mathbf{d_{si} \geq i}$, para todo **i** entre **1** y **k**, entonces el nuevo candidato es factible. Añadirlo a la solución
  - En otro caso, rechazar el candidato

Planificación de tareas

```
funcion Planificacion(int [0..n] d): int [0..n] s, k
    d[0] = 0   // centinela en el vector d deadlines
    s[0] = 0   // centinela en el vector s con la secuencia solucion
    k = 1       // k representa el número de tareas incluidas en la secuencia solución
    s[1] = 1   // la tarea con la mayor ganancia (la 1) siempre se incluye
    para  i = 2 hasta n     // Bucle greedy (voraz)
        r = k   // r tiene la posicion donde se ejecuta la nueva tarea
        mientras (d[s[r]] > d[i] and d[s[r]] ≠ r)   // mantener la solución ordenada crecientemente por d
            r = r - 1     // mover a la derecha las tareas siempre que queden dentro de su deadline
        si (d[s[r]] ≤ d[i] and d[i] > r)     // si están en orden las tareas y se puede incluir la tarea i
            para m = k hasta r + 1 decremento 1
                s[m+1] = s[m]
            s[r+1] = i   // coloca i en su lugar de ejecucion
            k = k + 1
    devolver s y k
```

Planificación de Tareas

- **Ejemplo**: n = 6

  b = (20, 15, 10, 7, 5, 3)

  d = (3, 1, 1, 3, 1, 3)

T	1	2	4
S	2	4	1
b	15	7	20
d	1	3	3

- **Idea**: suponer una solución óptima y comprobar que tiene el mismo beneficio que la calculada por el algoritmo

Planificación de tareas

- **Orden de complejidad** del algoritmo, suponiendo **n** tareas:
- Primero, ordenar las tareas por orden creciente de deadline (d_i): O(n log n)
- Repetir para i desde 1 hasta n:
  - Elegir el próximo candidato: O(1)
  - Comprobar si la nueva planificación es factible, y añadir la tarea a la solución en caso afirmativo: O(i) en el peor caso
- En total, el algoritmo es un $O(n^2)$

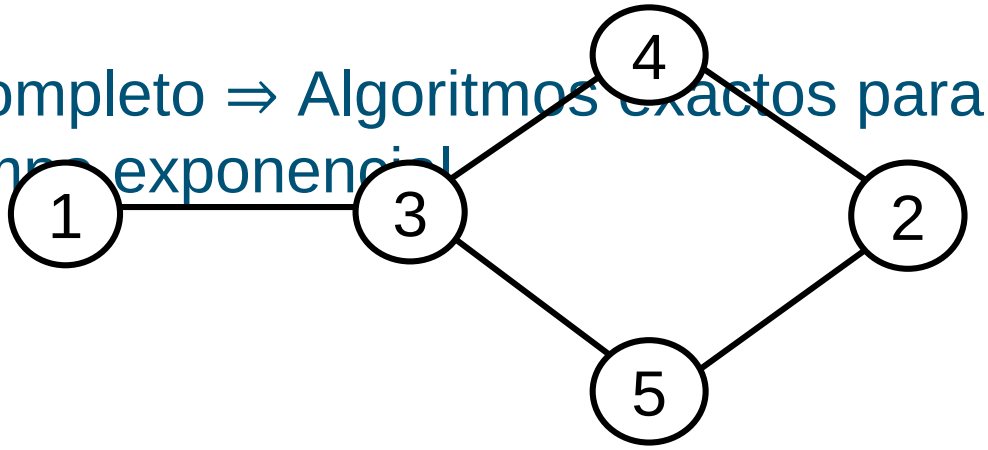
- Analizar **posible mejora** ⇒ en lugar de desplazar tareas, planificar cada tarea lo más tarde posible según su deadline y la ordenación actual

Planificación de tareas

- **Esquema greedy mejorado**
- Un conjunto de tareas J es factible si y solo si podemos construir una sucesión factible que incluya a las tareas en J como sigue: para cada tarea i ∈ J, ejecutar i (aún no seleccionada) en el instante t, donde t es el mayor entero que verifica que $0 \leq t \leq \min(n, d_i)$, donde n es el numero de tareas propuestas y d_i es la fecha límite (deadline) para la tarea i
- En otras palabras: añadir cada tarea i ∈ J a la sucesión que se está construyendo tan atrás como sea posible, pero no detrás de su fecha límite (deadline)
- Si una tarea no puede ejecutarse a tiempo por su deadline, entonces no se añade a J, pues J no será factible

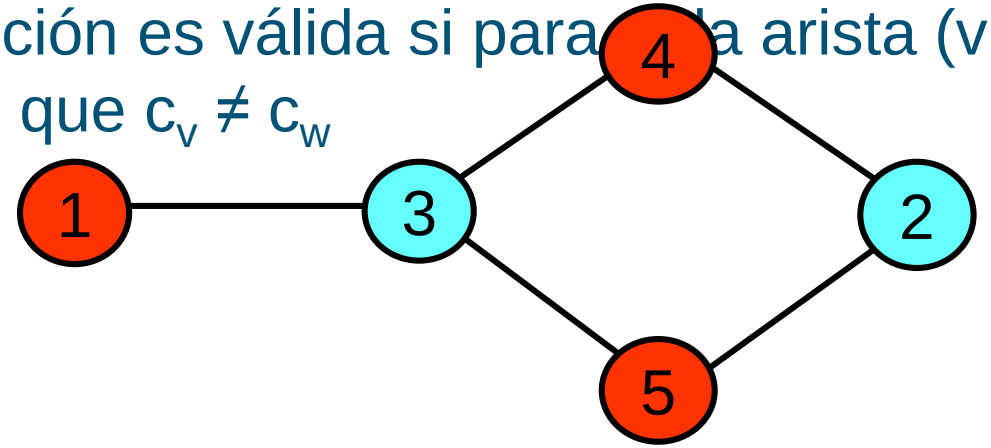
Coloreado de grafos

- **Coloreado de un grafo**: Dado un grafo no dirigido G = (V, E), asignar un color a cada vértice, de forma que dos vértices unidos por una arista tengan siempre distinto color
- **Problema del coloreado**: Dado un grafo no dirigido, realizar un coloreado utilizando el número mínimo de colores
- Problema NP-completo ⇒ Algoritmos exactos para este problema ⇒ tiempo exponencial



Coloreado de grafos

- **Planteamiento formal de la representación de la solución**: Una solución S tiene la forma (s_1, s_2, ..., s_n), donde s_i es un par (vértice, color), cumpliendo que para todo (v_i, c_i) y (v_j, c_j), si (v_i, v_j) es una arista del grafo, entonces se cumple que $c_i \neq c_j$
- La solución es válida si para toda arista (v, w) ∈ E, se cumple que $c_v \neq c_w$



- S = ((1, 1) (2, 2) (3, 2) (4, 1) (5, 1)) ⇒ Número de colores = 2

Coloreado de grafos

- Podemos usar una **heurística greedy** para obtener una solución (no es óptima):
- Inicialmente ningún vértice tiene color asignado
- Se toma un color colorActual = 1
- Para cada uno de los vértices sin colorear:
  - Comprobar si es posible asignarle el color actual
  - Si se puede, se asigna. En otro caso, se deja sin colorear
- Si quedan vértices sin colorear, escoger otro color (colorActual = colorActual + 1) y volver al paso anterior
- **Ejemplo**: asignación de días de exámenes (colores = días, asignaturas con alumnos en común = vértices adyacentes)

Coloreado de grafos

- La estructura básica del esquema voraz se repite varias veces, una por cada color, hasta que todos los vértices estén coloreados
- Función de selección: cualquier candidato restante
- Función Factible(v): se puede asignar un color a v (vértice actual) si ninguno de sus vértices adyacentes tiene ese mismo color

para todo vértice v adyacente a w **hacer**

si c_v == colorActual **entonces**

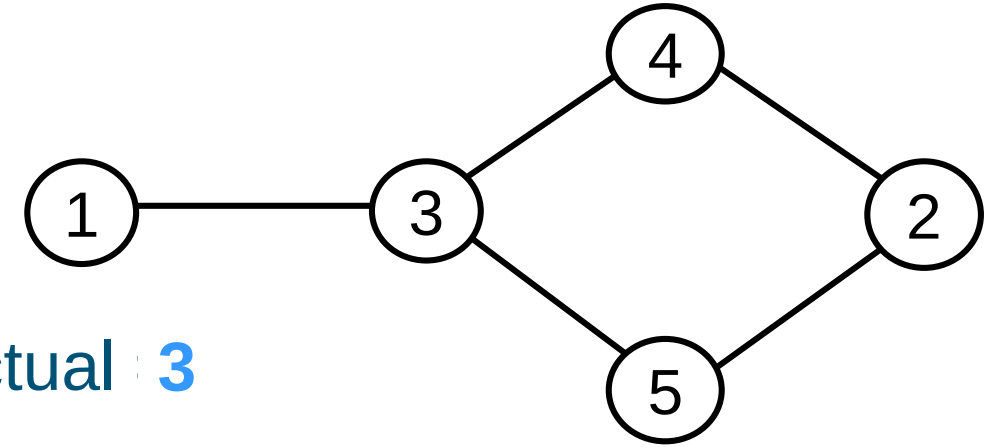
devolver false

devolver true

Coloreado de grafos

```
funcion coloreaGrafoGreedy(Grafo g)
    verticesNoColoreados = g.getVertices()  // lista de vértices del grafo
    mientras  !verticesNoColoreados.isEmpty() hasta     // Bucle greedy (voraz)
        u = verticesNoColoreados.getFirst()   // primer vértice no coloreado
        c = color no añadido aún
        u.colorea(c)
        para cada vértice v  ∈ verticesNoColoreados
            si (v no tiene ningún adyacente de color c)
                verticesNoColoreados.remove(v)
                v.colorea(c)
```

Coloreado de grafos



colorActual : **3**

c_1	c_2	c_3	c_4	c_5

- Resultado: se necesitan 3 colores. Recordar que el óptimo es 2
- Conclusión: el algoritmo no es óptimo
- ¿Cuál será el tiempo de ejecución en el peor caso?

Coloreado de grafos

- **Discusión de resultados**
- El algoritmo coloreaGrafoGreedy no es un algoritmo aproximado: no podemos poner cota al error de la solución obtenida ya que crece con el número de vértices del grafo
- Es un algoritmo heurístico, tiene la posibilidad, pero no la certeza, de encontrar la solución óptima. Por ejemplo, para un grafo bipartito de n vértices la solución óptima es 2 colores, pero el algoritmo puede llegar a utilizar n/2 colores
- No obstante, como todos los algoritmos exactos conocidos para el problema del coloreado de un grafo emplean un tiempo exponencial, es indudable la utilidad de la heurística voraz o greedy

Coloreado de grafos

- **Cálculo de eficiencia - Tiempo de ejecución**
- Sea n el número de vértices del grafo:
- Su eficiencia en el mejor caso es O(n)
- Su eficiencia en el peor caso es $O(n^3)$
  - Peor caso: grafo completo (todos los vértices están conectados entre sí)
  - El bucle externo se realiza n veces ya que a cada pasada sólo se va a lograr colorear un único nodo: O(n)
  - El bucle interno se realiza para todos los vértices no coloreados: O(n)
  - Dentro del bucle interno, para cada vértice, puede ser necesario comprobar sus n-1 vértices adyacente: O(n)

Códigos de Huffman

- Los **códigos de Huffman** son una técnica muy útil para comprimir archivos
- El algoritmo **greedy de Huffman** utiliza una tabla de frecuencias de aparición de cada carácter para construir una forma óptima de representar los caracteres con códigos binarios
- Ejemplo: Comprimir un archivo con 100000 caracteres, con la siguiente tabla de frecuencias

carácter	a	b	c	d	e	f
frec. (mil)	45	13	12	16	9	5

Códigos de Huffman

- **Solución 1**: Código de longitud fija de 3 bits (300000 bit

carácter	a	b	c	d	e	f
código	000	001	010	011	100	101

- **Solución 2**: **Códigos de Huffman**. Código de longitud variable dando codificaciones cortas a los caracteres más fre (224000 bit

carácter	a	b	c	d	e	f
código	0	101	100	111	1101	1100

Códigos de Huffman

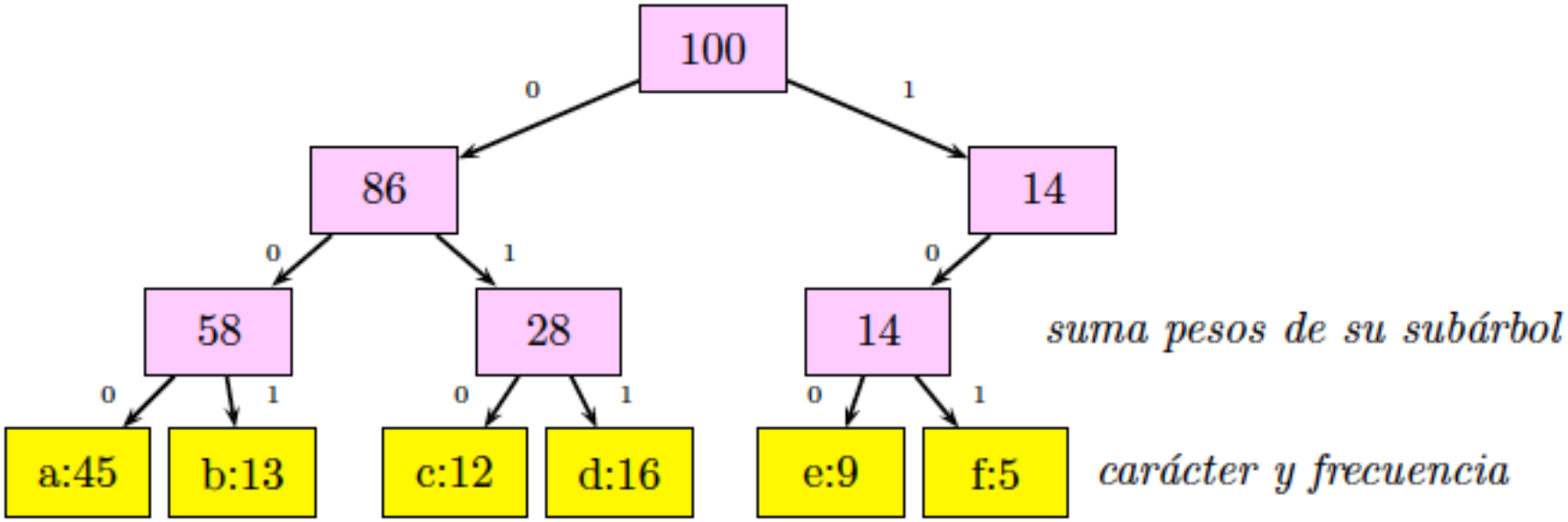
- **Algoritmo greedy de Huffman (1952)**
- Utiliza una **tabla de frecuencias** de aparición de cada carácter para construir una forma óptima de representar los caracteres con códigos binarios
- **Código de longitud variable**: codificaciones más cortas a los caracteres más frecuentes
- **Código libre de prefijos**:
  - Ninguna codificación puede ser prefijo de otra
  - Decodificación inmediata: “001011101” es “0 0 101 1101” = “aabe”

Códigos de Huffman

- **Algoritmo greedy de Huffman (1952)**
- Se representa mediante un **trie** (árbol lexicográfico):
  - Árbol binario cuyas **hojas** son los caracteres codificados
  - **Código de un carácter**: es el camino desde la raíz a la hoja que lo contiene, etiquetando con “0” las ramas de los hijos izquierdos, y con “1” hijos derechos
  - Cada **hoja** está etiquetada con un carácter y su frecuencia
  - Cada **nodo interno** está etiquetado con la suma de los pesos (frecuencias) de las hojas de su subárbol
  - Código óptimo para un alfabeto C. Siempre está representado por un **árbol lleno**. Cada nodo interno tiene 2 hijos. El árbol del código óptimo tiene |C| hojas y |C|- 1 nodos internos

Códigos de Huffman

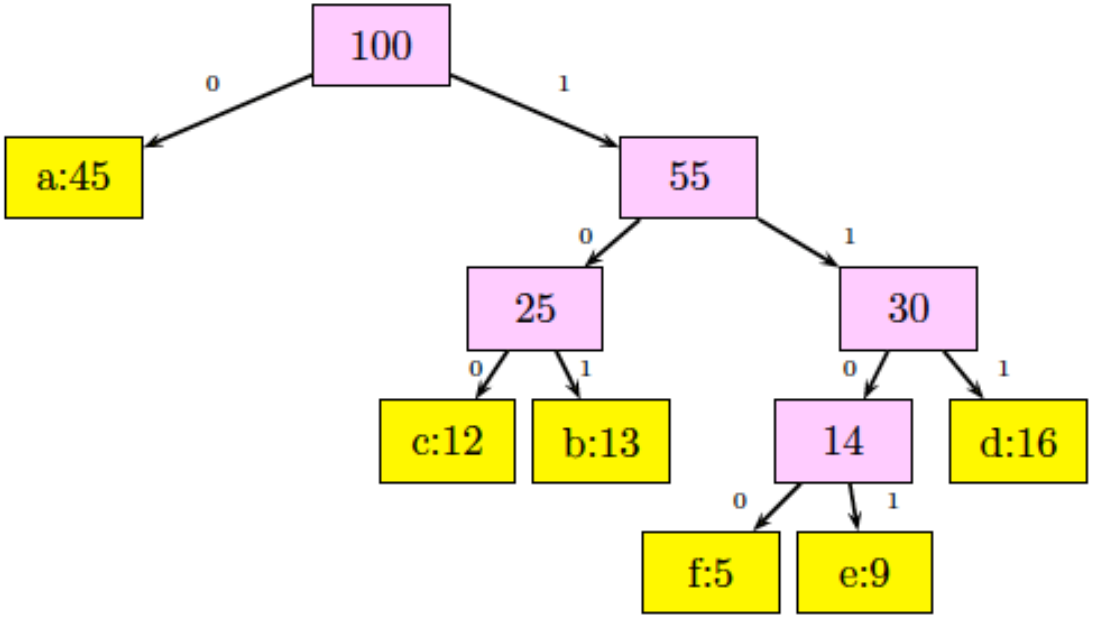
- Ejemplo: Árbol para el Código 1



carácter	a	b	c	d	e	f
código 1	000	001	010	011	100	101

Códigos de Huffman

- Ejemplo: Árbol para el Código 2. Código Huffman



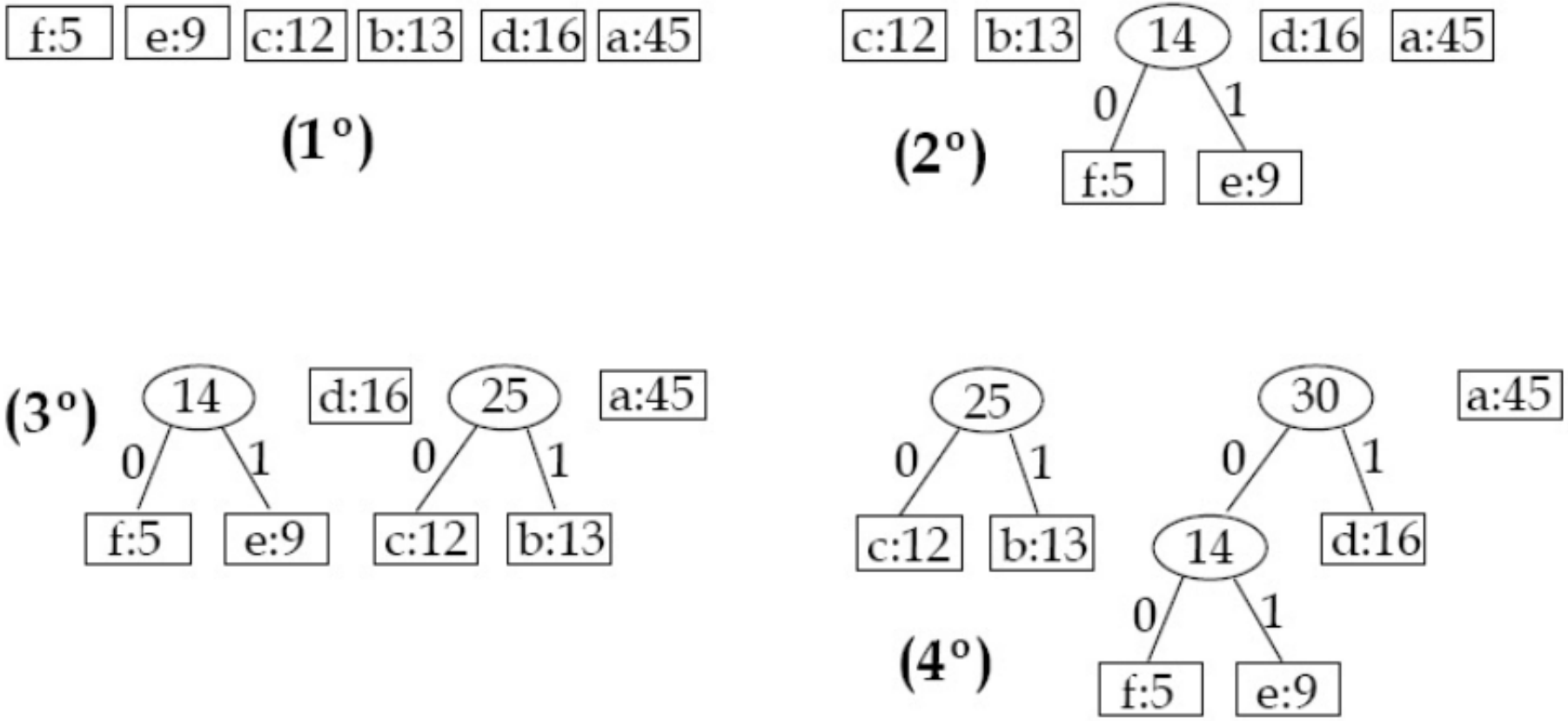
carácter	a	b	c	d	e	f
código 2	0	101	100	111	1101	1100

Códigos de Huffman

- **Algoritmo greedy de Huffman**
- Construye el árbol de un **código óptimo** de **abajo hacia arriba**
- **Q**: cola de prioridad de subárboles ordenados por frecuencia (las frecuencias hacen de prioridades)
- Inicialmente: conjunto de |C| hojas en Q
- Realiza una secuencia de |C|-1 mezclas hasta crear el árbol final
- En cada **iteración**: Se mezclan los dos subárboles de la cola Q que tienen menos frecuencia y resulta un nuevo subárbol cuya frecuencia (en el nodo raíz) es la suma de las frecuencias de los dos árboles mezclados

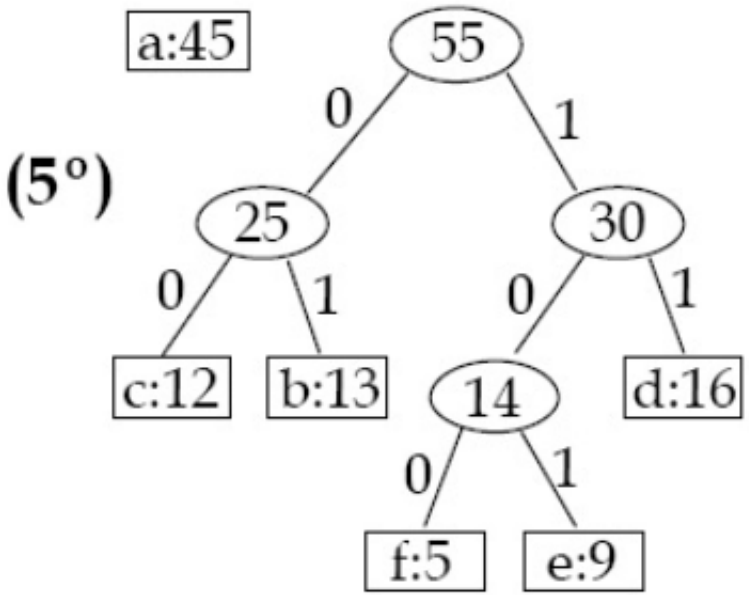
Códigos de Huffman

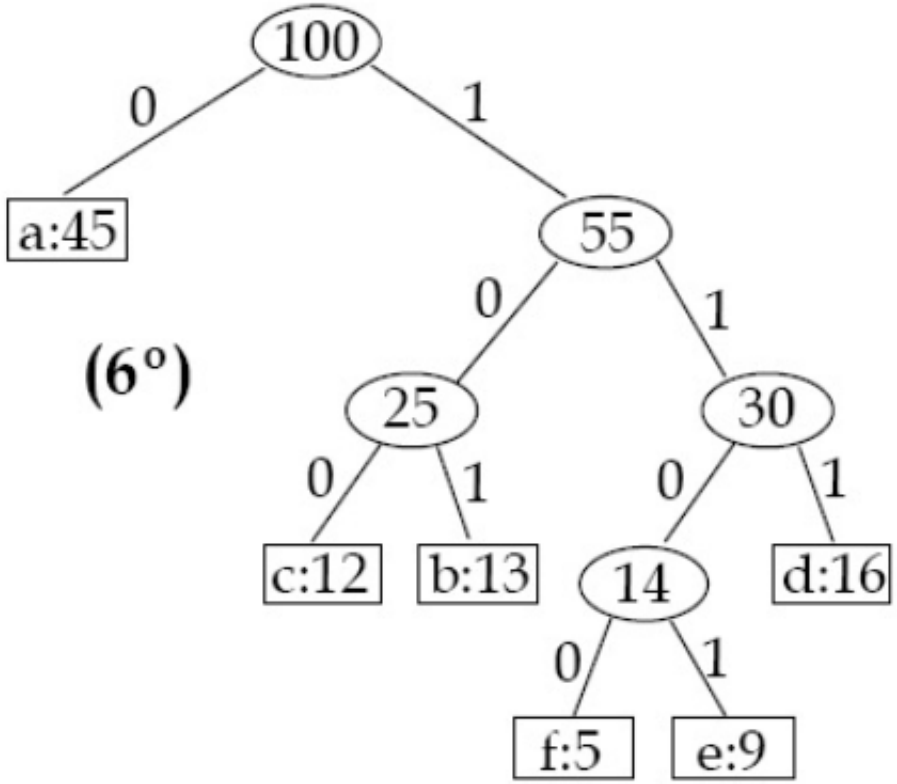
Ejemplo de construcción del árbol (I)



Códigos de Huffman

Ejemplo de construcción del árbol (II)





Códigos de Huffman

```
funcion Huffman(Alfabeto C) : árbol binario
    n = tamaño(C)
    Q. crear()   // cola de prioridad o heap minimo de árboles con criterio del minimo
    para todo c en C   // insertar en Q todo par (c, f(c)) - carácter y su frecuencia
        Q.insertar(c, f(c))
    para i = 1 hasta  n    // Bucle greedy (voraz)
        z = crearNodo()   // Crea un árbol de raíz z
        (x, f(x)) = Q.extraerMinimo()
        (y, f(y)) = Q.extraerMinimo()
        HijoIzq(z) = x
        HijoDer(z) = y
        f(z) = f(x) + f(y)
        Q.insertar((z, f(z)))
    (z, f(z)) = Q.extraerMinimo()
    devolver z
```

Códigos de Huffman

- **Algoritmo greedy de Huffman - Tiempo de ejecución**
- Alfabeto C con n símbolos ⇒ |C| = n
- Inicialización: O(n) (Inicializar la cola de prioridad - heap)
- Bucle se realiza n − 1 veces:
  - Reordenar el heap en cada iteración: ⇒ O(log n)
- Tiempo total: $T_{Huffman}(|C| = n) \in O(n \log n)$

Conclusiones sobre el esquema algorítmico Greedy

- **Greedy** se basa en una idea intuitiva:
  - Empezamos con una solución vacía, y vamos construyendo la solución al problema paso a paso
  - En cada paso se selecciona un candidato (el más prometedor) y se decide si se considera o no (función factible)
  - Una vez tomada una decisión no se vuelve a deshacer
  - Acabamos cuando tenemos una solución (que puede ser la óptima o no) o cuando no queden más candidatos
- **Cuestiones importantes**: ¿cuáles son los candidatos?, ¿cuáles son los candidatos más prometedores? ¿cómo se representa una solución al problema?, ¿cuál es la función de selección más adecuada?

Conclusiones sobre el esquema algorítmico Greedy

- Hay que diseñar una **función de selección** adecuada
  - Algunas pueden garantizar la solución óptima
  - Otras pueden ser más heurísticas
- **Función factible**: garantizar las restricciones del problema
- En general los **algoritmos greedy** son la solución rápida a muchos problemas (a veces obtenemos soluciones óptimas, otras no)
- ¿Y si podemos deshacer decisiones? ⇒ **Backtracking**

UNIVERSIDAD DE ALMERÍA