Tema 3. Algoritmos Voraces

Tabla de Contenidos

.:: Miguel Ángel Redondo ::.

Características generales

- Estrategia de descomposición de las soluciones
  - La solución está formada por “elementos” de solución
  - La solución es un conjunto de “elementos” de solución
- Método para obtener el conjunto solución
  - Estudiar cada “elemento”
  - Decidir si introducirlo o no en el conjunto
  - La decisión será irreversible
  - La solución se construye “paso a paso”
- Aplicación para problemas de optimización
  - Obtención de una buena solución sujeta a restricciones con una baja complejidad
    - Óptimo local
    - Óptimo global

Elementos a considerar

1. Conjunto de “elementos” candidatos a formar la solución
2. Conjunto de “elementos” ya seleccionados
3. Función solución
   - Decide si el conjunto de “elementos” seleccionados es o no solución al problema
4. Criterio de factibilidad
   - Decide si un conjunto de “elementos” candidatos nos puede llevar o no a una solución
5. Función selección
   - Toma el mejor “elemento” de los posibles candidatos
6. Función objetivo
   - Es lo que se trata de optimizar

Ejemplo de devolución de cambio

- Menor número de fracciones posibles {1,5,10,20,50}
  1. C = {1,5,10,20,50}
  2. S fracciones ya seleccionadas
  3. ¿Suma de los C igual a la cantidad a devolver?
  4. ¿Sobrepasa el total lo que llevamos hasta ahora más el nuevo elemento?
  5. Tomar siempre la fracción de mayor valor
  6. El número de fracciones a devolver ha de ser mínimo

Cumple las características de un problema solucionable mediante un algoritmo Voraz

Esquema de un algoritmo voraz

```
conjunto Voraz (conjunto C) {
    S ← ∅                  // inic. Solución es cjto. Vacio
    while (C) {
        x ← seleccionar(C);       // elemento a analizar
        C ← C – {x}               // decisión irreversible
        if (Factible(S,x)) S ← S U {x}
    }
    return (S);
}
```

Un escenario típico: “la mochila”

- Mochila de capacidad **M** (peso)
- Conjunto de ***n*** objetos, cada uno de ellos con un valor $\boldsymbol{v_i}$ y un peso $\boldsymbol{p_i}$
- Objetivo
  - Llenar la mochila maximizando el valor total de los objetos seleccionados
  - Se pueden considerar a los objetivos como fraccionables o no

Mochila con elementos fraccionables

```
void mochila(elemento_t c[MAX], float m, float s[MAX])
{
   int i;
   for (i=0; i<MAX; i++) s[i]=0;
   i=0;
   while ( (c[i].peso <= m) && (i < MAX) ) {
      s[i] = 1;
      m = m - c[i].peso;
      i++;
   }
   if (i<MAX) s[i] = m/c[i].peso;
}
```

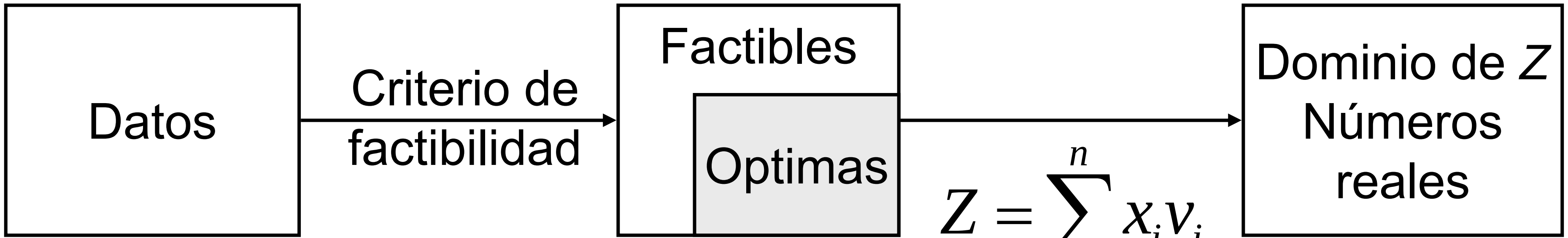
```
typedef struct {
  int valor, peso;
  char nombre[10];
  float densidad;
} elemento_t;
```

Descripción del método

Características del problema

- Datos del problema
  - Cantidad y forma
- Criterio de factibilidad
  - Indicar cómo son las soluciones a construir
  - Indicar las limitaciones existentes
  - SF conjunto de las soluciones factibles
  - OP $\subset$ SF, subconjunto de soluciones óptimas
- Función objetivo
  - Aplicada sobre el conjunto de las soluciones nos indicará cuáles son las soluciones óptimas

Esquema situación



$$Z = \sum_{i=1}^{n} x_i v_i$$

Demostración de la optimalidad

- Consideraciones iniciales
  - Función objetivo $\sum_{i=1}^{n} x_i v_i$
  - Restricción $\sum_{i=1}^{n} x_i p_i \leq M$
  - Factibilidad $\sum_{i=1}^{j} x_i p_i \leq M \quad \text{con} \quad x_i \in X$
- Tenemos que verificar que *X* produce mayor ganancia que cualquier otra solución factible

Demostración de la optimalidad

- Consideramos el vector de densidad ordenado de mayor a menor

$$D = \left[\frac{v_1}{p_1}, \frac{v_2}{p_2}, \ldots, \frac{v_n}{p_n}\right] \qquad d_i = \frac{v_i}{p_i}$$

- El algoritmo obtiene una solución

$$X = [1,1,\ldots,x_h,0,0,\ldots 0]$$

- Donde (por poder partir elementos)

$$0 \le x_h = \frac{M - \sum_{i=1}^{h-1} p_i}{p_h} \le 1 \qquad p_h \le M - \sum_{i=1}^{h-1} p_i$$

Demostración de la optimalidad

- Suponemos una solución factible *Y*

$$Y = [y_1, y_2, \ldots, y_n]$$

- Comprobamos que

$$\sum_{i=1}^{n} x_i v_i - \sum_{i=1}^{n} y_i v_i \geq 0$$

- Desarrollamos las expresiones y nos centramos en la parte que es diferente

Almacenamiento en cintas

- *N* Programas (o cjtos. de datos) para guardar en una cinta magnética donde deben entrar todos
- Cada programa tiene una longitud l_i
- Tiempo de acceso proporcional al lugar donde se almacena (*tiempo de búsqueda*)
- Objetivo: minimizar el tiempo medio de búsqueda

Datos del problema

- *n* Programas numerados de 1..n
- *n* longitudes L_i
- $L^*=(L_1,\ldots,L_n)$

Soluciones factibles

- Cualquier ordenación de L
- Permutación $L_i^* \equiv$ reordenación de L^*
- No hay restricción porque todos los programas caben en la cinta
  - o $L_i^*=(L_{i1},L_{i2},\ldots,L_{in})$

Almacenamiento en cintas

❑ Función objetivo

- Programa $L_{ij} \equiv L_j$ indica el lugar de L_{ij} en L_i^*
- Tiempo de acceso a L_{ij} proporcional a las longitudes de los programas colocados delante de él

$$T_{ij} = \sum_{k=1}^{j-1} L_{ik}$$

- Tiempo medio de acceso

$$Z(L_i^*) = \frac{1}{n}\sum_{j=1}^{n} T_{ij}$$

- Sustituimos T_{ij} por su valor

$$Z(L_i^*) = \frac{1}{n}\sum_{j=1}^{n}\sum_{k=1}^{j-1} L_{ik} = \frac{1}{n}\sum_{k=1}^{n-1}(n-k)L_{ik}$$

Almacenamiento en cintas

Objetivo que se persigue

- Obtener una solución factible L_i* tal que Z sea lo más pequeño posible

Mecanismo

- Elegir programas de longitud corta para valores grandes de *(n-k)* y viceversa
- Cada L_{ik} se suma *(n-k)* veces
- **Ordenar *L* de menor a mayor**

Calidad de la solución

- Solución óptima
- Es necesario demostrarlo

Calidad de la solución

- $L_i^+ \equiv$ solución obtenida (orden creciente de longitud)
- $L_i^* \equiv$ cualquier solución factible
- Comprobar que

$$Z(L_i^+) \le Z(L_i^*) \iff Z(L_i^+) - Z(L_i^*) \le 0$$

- Suponemos que L_i^* tiene elementos no ordenados de menor a mayor (sino sería L_i^+)
  - Elementos $L_{ia} > L_{ib}$ con $a<b$
- Intercambiamos los elementos de los lugares *a* y *b* y obtenemos otra permutación L_i^-
- Tendremos que ver que $Z(L_i^-)$ es menor que $Z(L_i^*)$

❑ Calidad de la solución (*cont.*)

$$Z(L_i^-) - Z(L_i^*)$$

$$\sum_{k\neq a,b}(n\text{-}k)L_{ik} + (n-a)L_{ib} + (n-b)L_{ia} - \sum_{k\neq a,b}(n\text{-}k)L_{ik} + (n-a)L_{ia} + (n-b)L_{ib}$$

$$(a-b)(L_{ia} - L_{ib}) < 0$$

- A medida que fuésemos detectando elementos desordenados y los fuésemos ordenando obtendríamos valores menores de ***Z***, acercándonos a ***Z(L$_i^+$)***
- Si todos los elementos están ordenados obtenemos el menor valor de ***Z*** que es ***Z(L$_i^+$)***

Camino mínimo de un vértice a otro

- Algoritmo de Dijkstra

Árbol de recubrimiento mínimo

- Recorrido del grafo cuyos vértices son los nodos del árbol y cuyos arcos tienen peso mínimo
- Algoritmo de Kruskal
  - Conjunto de aristas
  - Toma las aristas de menor peso
  - Verifica que no se formen ciclos
- Algoritmo de Prim
  - Conjunto de vértices visitados y no visitados
  - Toma la arista mejor (menor coste)
  - Verifique que no se formen ciclos

Algoritmo de Kruskal

Función Kruskal (G=(N,A):grafo; peso:A→R*):conjunto_aristas

{ Inicialización}

Ordenar el conjunto A según peso creciente

n ← #N

T ← ∅ *{mantiene las aristas del árbol de expansión}*

inicializar *n* conjuntos, cada uno con un elemento distinto de N

{ Bucle Voraz }

repetir

{u,v} ← la arista más corta aun no considerada

u_conj ← buscar(u)

v_conj ← buscar(v)

si u_conj ≠ v_conj **entonces**

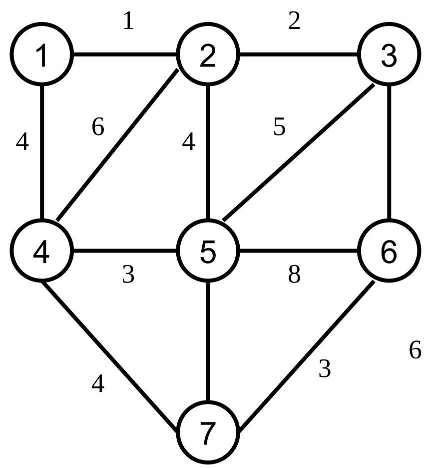
fusionar(u_conj,v_conj)

T ← T ∪ {{u,v}}

hasta #T = n-1

devolver T

Algoritmo de Kruskal



Etapa	*Arista*	*Componentes conexas*
inicialización		{1} {2} {3} {4} {5} {6} {7}

```
Función Kruskal (G=(N,A):grafo;
peso:A→R*):conjunto_aristas
  { Inicialización}
  Ordenar el conjunto A según peso creciente
  n ← #N
  T ← ∅ {mantiene las aristas del árbol de expansión}
   inicializar n conjuntos
  { Bucle Voraz }
  repetir
     {u,v} ← la arista más corta aun no considerada
     u_conj ← buscar(u)
     v_conj ← buscar(v)
     si u_conj ≠ v_conj entonces
         fusionar(u_conj,v_conj)
         T ← T ∪ {{u,v}}
   hasta #T = n-1
devolver T
```

Algoritmo de Prim

Función Prim (G=(N,A):grafo; peso:A→R*):conjunto_aristas

  { Inicialización}

  T ← ∅ *{mantiene las aristas del árbol de expansión}*

  B ← un elemento cualquier de N

  mientras B ≠ N **hacer**

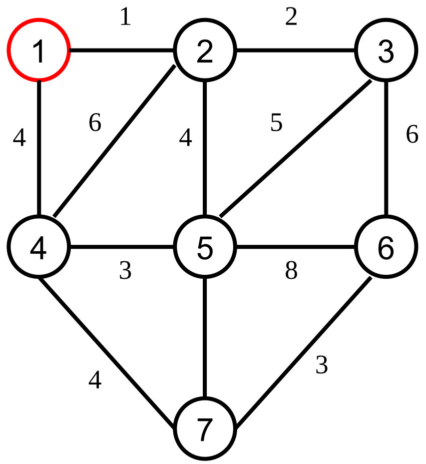
    encontrar una {u,v} de peso mínimo t.q. u∈N\B y v∈B

    T ← T ∪ {{u,v}}

    B ← B ∪ {u}

  devolver T

Algoritmo de Prim



Etapa	*{u,v}*	*B*
inicialización		{1}

Función Prim (G=(N,A):grafo; peso:A→R*):conjunto_aristas
{ Inicialización}
T ← ∅ *{mantiene las aristas del árbol de expansión}*
B ← un elemento cualquier de N
mientras B ≠ N **hacer**
encontrar una {u,v} de peso mínimo t.q. u∈N\B y v∈B
T ← T ∪ {{u,v}}
B ← B ∪ {u}
devolver T

T contiene las aristas seleccionadas: {2,1}, {3,2}, {4,1}, {5,4}, {7,4}, {6,7}

Problema

- Grafo dirigido y valorado G=(N,A)
- Dado un nodo (x_o) encontrar los mejores caminos (mínimos) que unen el resto de los nodos con el nodo x_o

Planteamiento

- Datos
  - Vértice inicial
  - Matriz de adyacencia (*L*) que define el grafo
    - *L[i,j]* es el valor de la arista *(i,j)*
    - *L[i,j]* = $\infty \Rightarrow$ no existe *(i,j)*
- Objetivo $\sum_{\mu_{0i}} D_{\mu_{0i}}$
- Criterio de factibilidad
  - μ_{0i} camino que une *Xo* con *Xi* (*Xo,…,Xi*)
  - $D_{\mu_{0i}}$ coste del camino
  - Cada para *Xij,Xik* consecutivo debe estar unido por un arco

Algoritmo de Dijkstra

Función Dijkstra (L[1..n,1..n]):vector[2..n]

{ Inicialización}

C ← {2,3,…,n} *{S=N \ C está implícito}*

para i ← 2 **hasta** n **hacer** D[i] ← L[1,i]

{Bucle Voraz}

repetir n-2 **veces**

v ← el elemento de C con D[v] mínimo

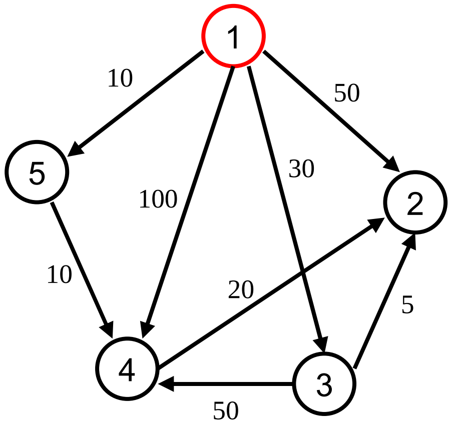
C ← C \ {v} *{implícitamente, S←S∪{v} }*

para cada elemento w de C **hacer**

D[w] ← min(D[w],D[v]+L[v,w])

devolver D (o T cjto. de aristas seleccionadas)

Algoritmo de Dijkstra



Etapa	{v}	C	D
inicialización		{2,3,4,5}	[50,30,100,10]

```
Función Dijkstra (L[1..n,1..n]):vector[2..n]
        { Inicialización}
        C ← {2,3,…,n} {S=N \ C está implícito}
        para j ← 2 hasta n hacer D[i] ← L[1,i]
        {Bucle Voraz}
        repetir n-2 veces
                v ← el elemento de C con D[v] mínimo
                C ← C \ {v}    {implícitamente, S←S∪{v} }
                para cada elemento w de C hacer
                        D[w] ← min(D[w],D[v]+L[v,w])
        devolver D
```

Algoritmos voraces

- Facilidad de aplicación
- Dificultad para demostrar que se alcanza una solución óptima

Heurística voraz

- Aplicación de algoritmo voraz para la obtención de buenas soluciones (no necesariamente óptimas)
  - Ilustra que los esquemas voraces no siempre conducen a una solución óptima
- Ejemplos
  - Coloreado de grafos
  - Viajante de comercio

Coloreado de grafos - Planteamiento

- Dado un grafo no dirigido G=(N,A)
- Se exige que todo par de vértices unidos por un arco tengan colores diferentes
- Emplear el menor número posible de colores
- Existen algoritmos que proporcionan solución óptima
  - Emplean un tiempo exponencial
  - No son adecuados para grafos de gran tamaño
  - Es más conveniente utilizar métodos aproximados

Coloreado de grafos – Algoritmo voraz

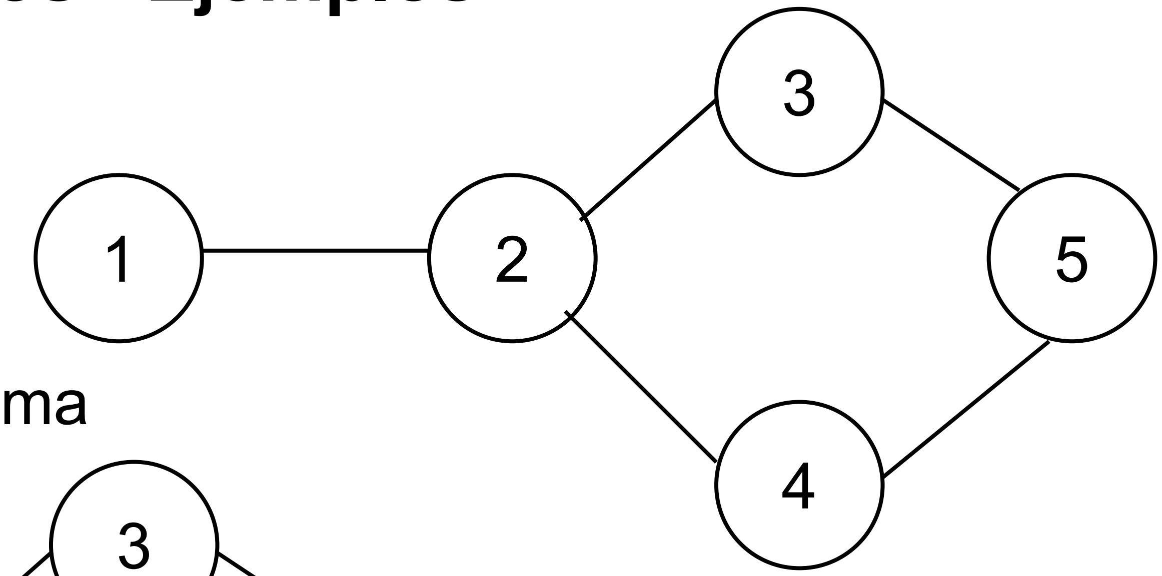
1. Repetir
   a. Elegir un color no elegido y un vértice no coloreado
   b. Asignar el color al mayor número de vértices respetando la restricción (adyacencia)
2. Hasta haber coloreado todos los vértices

Coloreado de grafos – Algoritmo voraz

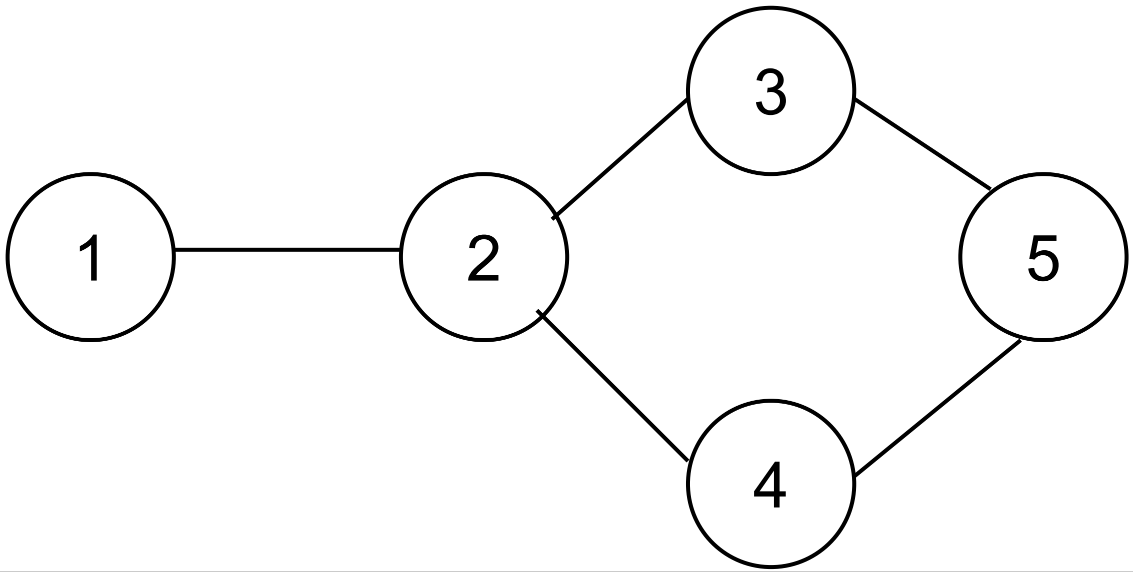
```
Algoritmo Coloreo (G(N, A));
   Inicializar_Colores;
   Mientras queden Nodos sin Colorear hacer
         c ← SiguienteColor;
         Para todo n∈N hacer
               Si Color(n)=SinColor
                     Si (No(c In Color(Vecinos(n)))
                           Asignar(n,c)
                     Fin_Si
               Fin_Si
         Fin_Para
   Fin_Mientras
Fin
```

Coloreado de grafos - Ejemplos

- Solución óptima



- Solución no óptima



Viajante de comercio - Planteamiento

- Dadas unas ciudades y distancias entre ellas
- Viajante debe partir de una de ellas y visitar el resto sólo una vez para finalmente volver al origen
- Debe recorrer la menor distancia posible
- Representación mediante un grafo completo no dirigido
  - Vértices = Ciudades, Arcos y pesos = distancias
- Existen algoritmos que proporcionan solución óptima
  - Emplean un tiempo exponencial
  - No son adecuados para grafos de gran tamaño
  - Es más conveniente utilizar métodos aproximados

Viajante de comercio – Algoritmo voraz

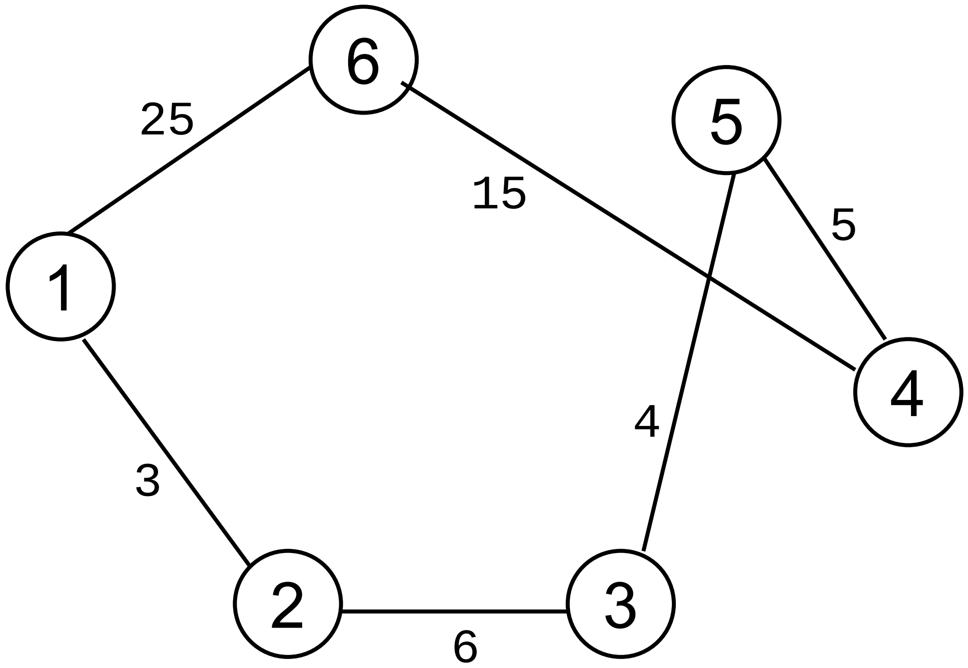
1. Repetir
    a. Seleccionar la arista más corta que cumpla
        i. Todavía no ha sido considerada
        ii. No forma ciclo con las aristas ya seleccionadas (en la última no se cumplirá esta restricción)
        iii. No es la tercera arista que incide en el mismo vértice de entre los seleccionados
2. Hasta haber visitado todos los vértices

Viajante de comercio – Ejemplo

Matriz de adyacencia

	2	3	4	5	6
1	$\boxed{3}$	10	11	7	$\boxed{25}$
2		$\boxed{6}$	12	8	26
3			9	$\boxed{4}$	20
4				$\boxed{5}$	$\boxed{15}$
5					18

Coste del camino = 58



Camino óptimo ⇒ coste= 56