Modelado y Solución para Problemas Combinatorios

Maestría en Ciencias de la Computación
Maestría en TICs

Facultad Politécnica
Universidad Nacional de Asunción

Julio, Agosto – 2015

Laura Ingolotti

Scheduling
Introducción

Scheduling es un proceso de toma de decisiones utilizado regularmente en la industria de servicios o de fabricación.

Tiene que ver con la asignación de recursos a tareas durante un determinado periodo de tiempo, de tal forma que se optimice uno o más objetivos.

Ejemplo de recursos: máquinas en una fábrica, pistas de aterrizaje en un aeropuerto, obreros en una construcción, vías en una línea ferroviaria, etc.

Scheduling - Notación
Modelos determinísticos

- Trabajos. El número de trabajos es finito (j).
- Máquinas. El número de máquinas es finito (i).
- Tareas: procesamiento de un trabajo en una máquina (i,j).
- Tiempo de procesamiento (p_{ij}). Tiempo empleado para procesar el trabajo j en la máquina i.
- Instante de inicio (r_j). Instante más temprano en el que un trabajo está listo para iniciar su procesamiento.
- Instante de finalización (d_j): Instante en el que debería finalizar un trabajo. Si un trabajo finaliza más tarde, se considera que hubo un atraso.
- Peso (w_j): prioridad asignada a un trabajo con respecto al resto de trabajos que forman parte del sistema.

Scheduling – Notación - Terminología
Modelos determinísticos

- Un problema de scheduling es definido por la tripleta $<\alpha, \beta, \gamma>$
- El elemento α describe el ambiente de máquinas y contiene una sola entrada. Posibles valores:
  - Una sola máquina (1).
  - Máquinas idénticas en paralelo (P_m).
  - Máquinas en paralelo con diferentes velocidades (Q_m).
  - Flow Shop: m máquinas en serie, cada trabajo tiene que ser procesado por cada una de las m máquinas. Todos los trabajos siguen la misma ruta (F_m).
  - Flexible Flow Shop (FF_c): cada trabajo debe ser procesado en diferentes etapas. Todos los trabajos siguen la misma ruta en cuanto a etapas se refiere. Cada etapa está formada por un conjunto de máquinas en paralelo. Cada trabajo puede elegir cualquiera de las máquinas que forman parte de la etapa en la que se encuentra.

Scheduling – Notación - Terminología
Modelos determinísticos

- … más atributos del elemento α

  - Job Shop (J_m): considerando m máquinas, cada trabajo tiene su propia ruta.

  - Open Shop (O_m): cada trabajo debe ser procesado por cada una de las m máquinas. Sin embargo, no hay restricción sobre la ruta de cada trabajo. Incluso, puede ser determinada por el scheduler.

Scheduling – Notación
Modelos determinísticos

- El campo β incluye restricciones de procesamiento tales como:
  - Instante de inicio (r_j).
  - Procesamiento continuo (sí o no).
  - Restricciones de precedencia.
  - Tiempo de configuración: tiempo entre el fin de procesamiento de un trabajo en una máquina y el inicio de procesamiento de otro trabajo en la misma máquina.
  - Familias de trabajo: el tiempo de configuración puede ser configurado por familia de trabajos.
  - Procesamientos por lote: cantidad de trabajos que pueden ser procesados por una misma máquina al mismo tiempo.

Scheduling – Notación
Modelos determinísticos

- Disponibilidad de las máquinas.
- Elegibilidad de las máquinas. Puede ocurrir que no todas las máquinas puedan procesar todos los trabajos.
- Orden de procesamiento de trabajos por máquinas en un Flow Shop.
- Bloqueo de máquinas. Una máquina no libera un trabajo cuando su procesamiento ha finalizado y por lo tanto, no recibe un siguiente trabajo.
- El trabajo no admite tiempo de espera. Se deberá indicar un tiempo de inicio del trabajo, el cual deberá permitir el procesamiento continuo del trabajo en las diferentes máquinas de su ruta.
- Recirculación: un trabajo puede visitar una misma máquina más de una vez.

Scheduling – Notación – Terminología
Función objetivo

Tiempo de finalización de un trabajo (C_j): Instante en el que finaliza el procesamiento de un trabajo en la última máquina de su ruta.

Demora de un trabajo: $C_j - d_j$

El campo γ se refiere a la función objetivo que debe ser minimizada. Ejemplos de funciones objetivo a ser minimizadas:

- Makespan ($\max C_j$): máximo instante de finalización, considerando el instante de finalización de cada trabajo del sistema.
- Máxima demora (considerando la demora de cada trabajo en el sistema).
- Suma ponderada de tiempos de finalización.

Scheduling
Modelos Estocásticos

Consideran la probabilidad de que ocurra un evento no esperado, lo cual puede modificar el tiempo de procesamiento de un trabajo, o la disponibilidad de una máquina, el tiempo de configuración entre el inicio de dos trabajos, etc.

Métodos de solución
Optimización Exacta – Programación Dinámica

Jobs	1	2	3
p_j	4	3	6
h_jC_j	$C_1 + C_1^2$	$3+C_2^3$	$8*C_3$

Condición Inicial:

V{j} = hjCj para cada trabajo j que pertenezca al conjunto de trabajos J = {1, 2, 3}

Relación recursiva: $$V(J) = \min_{j \in J} \left(V(J - \{j\}) + h_j \sum_{k \in J} p_k \right)$$

Primera Iteración: Se trata el problema como si estuviera formado por un solo trabajo. Se calcula su función objetivo para cada posible instancia del nuevo problema.

V{1} = C1 + C12 = 4 + 16 = 20
V{2} = 3 + C23 = 3 + 27 = 30
V{3} = 8*C3 = 8 * 6 = 48

Ejemplo: Programación Dinámica

Segunda Iteración: Se agrega un trabajo más. En esta iteración se considera que el problema está formado por 2 trabajos.

Instancia 1: J = {1, 2}

$V(\{1,2\}) = V(1) + h_2C_2 = 20 + 3+C_2^3 = 20 + 3 + (3 + 4)^3 = 20 + 3 + 343 = 366$

$V(\{2,1\}) = V(2) + h_1C_1 = 30 + C_1 + C_1^2 = 30 + (4 + 3) + (4 + 3)^2 = 30 + 7 + 49 = 86$

$\min(V(\{1,2\}), V(\{2,1\})) = V(\{2,1\})$

Instancia 2: J = {1, 3}

$V(\{1,3\}) = V(1) + h_3C_3 = 20 + 8*C_3 = 20 + 8* (4 + 6) = 20 + 80 = 100$

$V(\{3,1\}) = V(3) + h_1C_1 = 48 + C_1 + C_1^2 = 48 + (6 + 4) + (6 + 4)^2 = 48 + 10 + 100 = 158$

$\min(V(\{1,3\}), V(\{3,1\})) = V(\{1,3\})$

Ejemplo: Programación Dinámica

Instancia 3: J = {2, 3}

$V(\{2,3\}) = V(2) + h_3C_3 = 30 + 8*C_3 = 30 + 8*(3 + 6) = 30 + 72 = 102$

$V(\{3,2\}) = V(3) + h_2C_2 = 48 + 3 + C_2^3 = 48 + 3 + (6 + 3)^3 = 48 + 3 + 729 = 780$

$\min(V(\{2,3\}), V(\{3,2\})) = V(\{2,3\})$

Tercera Iteración: Se agrega un trabajo más. Se obtiene el problema original.

Instancia 1: J = {1, 2, 3}

$V(\{2,1,3\}) = V(\{2,1\}) + h_3C_3 = 86 + 8*C_3 = 86 + 8*(3 + 4 + 6) = 86 + 104 = 190$

$V(\{1,3,2\}) = V(\{1,3\}) + h_2C_2 = 100 + 3 + C_2^3 = 100 + 3 + (4 + 6 + 3)^3 = 100 + 3 + 2197 = 2300$

$V(\{2,3,1\}) = V(\{2,3\}) + h_1C_1 = 102 + C_1 + C_1^2 = 102 + (3 + 6 + 4) + (3 + 6 + 4)^2 = 102 + 13 + 169 = 284$

Secuencia óptima = 2, 1, 3

Métodos de solución heurísticos
Procedimientos de propósito general

Son procedimientos cuyos pasos están definidos independientemente del dominio del problema a resolver.

Son técnicas heurísticas, no aseguran una solución óptima, pero garantizan un tiempo de cómputo manejable para instancias complejas de un problema.

Reglas de secuenciación

Reglas para definir el orden de procesamiento de las tareas.

Existe una clasificación que agrupa estas reglas en estáticas y dinámicas.

Reglas estáticas: no dependen del instante de tiempo en el que se lleva a cabo una acción.

Reglas de secuenciación – Estáticas y Dinámicas

- Reglas dinámicas: sus decisiones dependen del valor de determinadas variables en un instante de tiempo.

  Ejemplo: Minimum slack first. Cada vez que se debe tomar una decisión se ordenan los trabajos de forma ascendente, según el valor de la siguiente expresión:

Holgura:

$$\max(d_j - (t + p_j), 0)$$

Reglas de secuenciación – Locales y Globales

- Reglas locales: solo tienen en cuenta el estado de la máquina donde se va a realizar el procesamiento, o la cola de trabajos donde se debe tomar una decisión.
- Reglas globales: considera el estado de todo el sistema.

Reglas de secuenciación compuestas

Cuando un problema debe optimizar un único criterio, las reglas de secuenciación simple pueden ser una buena guía para llegar a una solución óptima o cercana a la óptima. Sin embargo, cuando diferentes criterios deben ser considerados al buscar una buena solución, se necesita una combinación de reglas de secuenciación, que no solo tengan en cuenta un aspecto del scheduling.

Ejemplo: Job Shop Scheduling

```
/* JSSP, Job-Shop Scheduling Problem */

/* Written in GNU MathProg by Andrew Makhorin <mao@gnu.org> */


/* Problema: conjunto de trabajos, donde cada uno debe operar sobre una determinada secuencia
de máquinas. Se desea obtener la secuencia de trabajos sobre cada máquina, tal que el máximo
tiempo de finalización, considerando todos los trabajos, sea mínimo. Cada máquina puede ser
utilizado por un solo trabajo a la vez, sin interrupción. Referencia: D. Applegate and W. Cook, "A
Computational Study of the Job-Shop Scheduling Problem", ORSA J. On Comput., Vol. 3, No. 2,
Spring 1991, pp. 149-156. */

/* número de trabajos */
param n, integer, > 0;

/* número de máquinas */
param m, integer, > 0;

/* conjunto de trabajos */
set J := 1..n;

/* conjunto de máquinas */
set M := 1..m;
```

Ejemplo: Job Shop Scheduling

/* parámetro que indica el orden en el cual cada trabajo debe utilizar las máquinas del sistema. */

```
param sigma{j in J, t in 1..m}, in M;

/* un mismo trabajo no puede pasar por una misma máquina dos veces */
check{j in J, t1 in 1..m, t2 in 1..m: t1 <> t2}:
     sigma[j,t1] != sigma[j,t2];

/* tiempo de procesamiento del trabajo j en la máquina a */
param p{j in J, a in M}, >= 0;

/* instante en el que el trabajo j inicia su procesamiento en la máquina a */
var x{j in J, a in M}, >= 0;

/* el procesamiento del trabajo j debe seguir el orden descrito en el parámetro sigma. */

s.t. ord{j in J, t in 2..m}:

     x[j, sigma[j,t]] >= x[j, sigma[j,t-1]] + p[j, sigma[j,t-1]];

/* La condición disyuntiva: cada máquina puede procesar como máximo un trabajo a la vez puede
ser descrita como:

     x[i,a] >= x[j,a] + p[j,a]  OR  x[j,a] >= x[i,a] + p[i,a]

  Para toda i, j in J, a in M. Esta condición es modelada utilizando las variables binarias Y, según se
muestra a continuación. */
```

Ejemplo: Job Shop Scheduling

```
/* Y[i,j,a] es 1 si i es procesado antes que j en la máquina a, y 0 si j es procesado antes que i en la
máquina a */

var Y{i in J, j in J, a in M}, binary;

/* se calcula el valor de una constante lo suficientemente grande */
param K := sum{j in J, a in M} p[j,a];

display K;

s.t. phi{i in J, j in J, a in M: i <> j}:
     x[i,a] >= x[j,a] + p[j,a] - K * Y[i,j,a];

s.t. psi{i in J, j in J, a in M: i <> j}:
     x[j,a] >= x[i,a] + p[i,a] - K * (1 - Y[i,j,a]);

var z;

/* se calcula el makespan, el cual es el máximo tiempo de finalización considerando los trabajos del
problema */

s.t. fin{j in J}: z >= x[j, sigma[j,m]] + p[j, sigma[j,m]];

/* función objetivo: instante de finalización del último trabajo que finalice sea el mínimo posible */

minimize obj: z;
```

Ejemplo: Job Shop Scheduling

```
/* Los datos corresponden a la instancia ft06 (mt06) from: H. Fisher, G.L. Thompson (1963),
Probabilistic learning combinations of local job-shop scheduling rules, J.F. Muth, G.L. Thompson
(eds.), Industrial Scheduling, Prentice Hall, Englewood Cliffs, New Jersey, 225-251 */

/* La solución óptima es 55 */

data;

param n := 6;
param m := 6;

param sigma :  1  2  3  4  5  6 :=
        1    3  1  2  4  6  5
        2    2  3  5  6  1  4
        3    3  4  6  1  2  5
        4    2  1  3  4  5  6
        5    3  2  5  6  1  4
        6    2  4  6  1  5  3 ;

param p     :  1  2  3  4  5  6 :=
        1    3  6  1  7  6  3
        2   10  8  5  4 10 10
        3    9  1  5  4  7  8
        4    5  5  5  3  8  9
        5    3  3  9  1  5  4
        6   10  3  1  3  4  9 ;
end;
```

Ejemplo: Bin Packing Problem (parte 1)

```
/* BPP, Bin Packing Problem */

/* Written in GNU MathProg by Andrew Makhorin <mao@gnu.org> */

/* Dado un conjunto de items I = {1,...,m} donde cada ítem i ocupa w[i] > 0, el problema Bin
Packing (BPP) consiste en guardar los ítems dentro de contenedores que tienen una capacidad igual a c, de tal
forma que se utilice el mínimo número de contenedores. */

/* cantidad de ítems */
param m, integer, > 0;


/* conjunto que define los identificadores de los ítems */
set I := 1..m;

/* el elemento w[i] determina el espacio que ocupa el ítem i */
param w{i in 1..m}, > 0;

/* capacidad del contenedor */
param c, > 0;


/* Para resolver este problema es necesario estimar una cantidad de contenedores como límite superior a la
cantidad necesaria para almacenar todos los ítems. Por ejemplo el número m podría ser utilizado, pero el autor de
este trabajo propone otra forma de estimar tal cantidad, que considera es más eficiente.

Propone poner los ítems en un contenedor mientras sea posible, y si el contenedor está lleno, usar otro bin. El
valor obtenido sería el límite superior de la cantidad de bins necesarios. La solución debe devolver una cantidad
que sea menor o igual al límite superior. */
```

Ejemplo: Bin Packing Problem (parte 2)

```
param z{i in I, j in 1..m} :=

/* z[i,j] = 1 si el ítem i está dentro del contenedor j, sino z[i,j] = 0 */

   /* poner el ítem 1 en el contenedor 1 */
   if i = 1 and j = 1 then 1

   /* si el ítem i ya está en algún contenedor, no ponerlo en otro contenedor j */

   else if exists{jj in 1..j-1} z[i,jj] then 0

   /* si el ítem i no cabe en el contenedor j, no ponerlo en dicho contenedor */
   else if sum{ii in 1..i-1} w[ii] * z[ii,j] + w[i] > c then 0

   /* si no cumple ninguna de las condiciones anteriores ponerlo en el contenedor j */
   else 1;

  /* cada ítem debe ser guardado en un solo contenedor */
  check{i in I}: sum{j in 1..m} z[i,j] = 1;

  /* la capacidad de ningún contenedor debe ser excedida*/
  check{j in 1..m}: sum{i in I} w[i] * z[i,j] <= c;

/* determinar el número de contenedores usados por la heurística; el cual es un límite superior de la solución
óptima*/

  param n := sum{j in 1..m} if exists{i in I} z[i,j] then 1;
  display n;
```

Ejemplo: Bin Packing Problem (parte 3)

```
/* identificadores de los contenedores */
set J := 1..n;

/* x[i,j] = 1 significa que el ítem i será guardado en el contenedor j */
var x{i in I, j in J}, binary;

/* used[j] = 1 significa que el contenedor j contiene al menos un ítem */

var used{j in J}, binary;

/* cada ítem debe estar guardado en un solo contenedor */
s.t. one{i in I}: sum{j in J} x[i,j] = 1;

/* si contenedor j es usado, su capacidad no debe ser excedida */
s.t. lim{j in J}: sum{i in I} w[i] * x[i,j] <= c * used[j];

/* minimizar la cantidad de contenedores utilizados*/
minimize obj: sum{j in J} used[j];

data;

param m := 6;

param w := 1 50, 2 60, 3 30, 4 70, 5 50, 6 40;

param c := 100;

end;
```

Ejemplo: Problema de Asignación (parte 1)

```
/* ASSIGN, Problema de Asignación *//* Written in GNU MathProg by Andrew Makhorin <mao@gnu.org> */

/* El problema de asignación es uno de los problemas de optimización combinatoria fundamentales. En su
forma más general puede ser definido como: “Considere un número de agentes y de tareas. Cualquier
agente puede realizar cualquier tarea. El coste de llevar a cabo tarea puede variar dependiendo del
agente que la realice. Cada tarea debe ser realizada por un único agente. Se desea encontrar una
asignación tarea-agente tal que el coste total sea mínimo. */

/* número de agentes */
param m, integer, > 0;

/* número de tareas */
param n, integer, > 0;

/* constantes que identifican a cada agente */
set I := 1..m;

/* constantes que identifican cada tarea */
set J := 1..n;

/* coste de que la tarea j sea realizada por el agente i */
param c{i in I, j in J}, >= 0;

/* x[i,j] = 1 significa que el agente i realiza la tarea j
var x{i in I, j in J}, >= 0;
```

Ejemplo: Problema de Asignación (parte 2)

```
/* cada agente puede llevar a cabo como máximo 1 tarea*/

s.t. phi{i in I}: sum{j in J} x[i,j] <= 1;

/* cada tarea debe ser asignada exactamente a un agente */
s.t. psi{j in J}: sum{i in I} x[i,j] = 1;

/* El objetivo es encontrar la asignación más barata */
minimize obj: sum{i in I, j in J} c[i,j] * x[i,j];

solve;

printf "\n";
printf "Agente  Tarea       Coste\n";
printf{i in I} "%5d %5d %10g\n", i, sum{j in J} j * x[i,j],
   sum{j in J} c[i,j] * x[i,j];
printf "---------------------\n";
printf "      Total: %10g\n", sum{i in I, j in J} c[i,j] * x[i,j];
printf "\n";
```

Ejemplo: Problema de Asignación (parte 3)

```
data;

/* These data correspond to an example from [Christofides]. */

/* Optimal solution is 76 */

param m := 8;

param n := 8;

param c : 1  2  3  4  5  6  7  8 :=
      1  13 21 20 12  8 26 22 11
      2  12 36 25 41 40 11  4  8
      3  35 32 13 36 26 21 13 37
      4  34 54  7  8 12 22 11 40
      5  21  6 45 18 24 34 12 48
      6  42 19 39 15 14 16 28 46
      7  16 34 38  3 34 40 22 24
      8  26 20  5 17 45 31 37 43 ;

end;
```

Algoritmo genético

Genetic Algorithms (GAs) son algoritmos de búsqueda que utilizan heurísticas adaptativas. Está basado en ideas del proceso evolutivo relacionadas con la selección natural y la herencia genética.

Fue desarrollado inicialmente por John Holland en 1975.

Realizan una exploración inteligente, aunque aleatoria, GAs exploran el espacio de búsqueda que mejor performance ha mostrado durante las iteraciones, por tener en cuenta información histórica que va considerando a lo largo de las iteraciones de la búsqueda.

GAs simulan la supervivencia de los individuos más aptos entre todos los individuos de una población, a través de las consecutivas generaciones que se construyen para resolver un problema de optimización.

Cada generación está formada por una población, la cual a su vez está formada por un conjunto de individuos. Cada individuo es un cromosoma y representa una posible solución dentro del espacio de búsqueda.

Los individuos en una población compiten por recursos y parejas. Los individuos más exitosos generarán mayor descendencia.

Genes de buenos progenitores se propagarán a través de la población, tal que buenos padres producirán con mayor probabilidad hijos que sean aún mejores que sus padres.

Así, cada generación se volverá más apta para su ambiente.

Algoritmos genéticos - Representación

Una población de individuos es mantenida dentro del espacio de búsqueda, cada uno representando una posible solución para un determinado problema.

Cada individuo es codificado como un vector de longitud finita, o como un conjunto de variables, en términos de algún alfabeto, usualmente el alfabeto binario {0,1}.

Para continuar con la analogía de procesos evolutivos, cada individuo se considera un cromosoma y cada elemento que define un individuo un gen. Así, un cromosoma (solución) está compuesto de varios genes (variables). El valor del fitness que es asignado a cada individuo representa la habilidad del individuo para competir en su ambiente.

El GA utiliza el valor de fitness de cada individuo para realizar la selección y posterior combinación de individuos, en el momento de construir la siguiente generación.

El GA mantiene una población de N cromosomas con sus correspondientes valores de fitness. Padres son seleccionados para formar parejas, basados en sus fitness. Las parejas producen descendientes mediante un método de cruce elegido. A los cromosomas con mejores valores de fitness se les da mayor probabilidad de ser seleccionados y por ende, de reproducirse.

Cada nueva generación contendrá mejores soluciones parciales que las anteriores generaciones. Se dice que un GA converge cuando la diferencia entre el fitness promedio de las nuevas generaciones con respecto a las anteriores es casi nulo.

Algoritmos genéticos – Detalles de Implementación

La población de la primera generación es generada de forma aleatoria.

Para hacer que cada población evolucione y se formen las siguientes generaciones, se utilizan tres operadores:

1. SELECCIÓN: este operador se basa en el fitness de cada individuo y debería asegurar la supervivencia del más fuerte.
2. CRUCE: este operador genera un nuevo individuo mediante la combinación de genes de los dos padres seleccionados. La forma de realizar la combinación depende del método de cruce que se selecciona.
3. MUTACIÓN: este operador modifica alguno de los genes de los descendientes.

Algoritmos genéticos – Operadores (1)

OPERADOR DE SELECCIÓN

Permite que los genes de los mejores individuos de una población puedan ser transmitidos a un individuo de la siguiente generación.

Un individuo es mejor o peor que otro, según el valor de fitness que se le asocia.

El valor de fitness se calcula evaluando la función objetivo del problema, la cual estará basada en determinidas características de los genes del individuo.

OPERADOR DE CRUCE

Dos individuos son elegidos de la población usando el operador de SELECCIÓN. Existen varios algoritmos para llevar a cabo el cruce de dos cromosomas. Según las características del problema, un algoritmo puede ser más apropiado que otro.

Un subconjunto de genes de cada individuo es aleatoriamente seleccionado e intercambiado para dar lugar al nuevo descendiente. El cruce se realiza con una probabilidad de cruce, la cual es un parámetro del algoritmo genético.

En un algoritmo de cruce de un solo punto, si S1=000000 y S2=111111 y el punto de cruce es 2, entonces S1'=110000 and S2'=001111. Los siguientes dos descendientes de este cruce son puestos en la siguiente generación de la población. En algunos casos el reemplazo es condicional, es decir no necesariamente los descendientes reemplazan a los padres. Son variantes del algoritmo genético que dependen de las características del problema que se esté intentando resolver.

Algoritmos genéticos – Operadores (2)

OPERADOR MUTACIÓN

Con una probabilidad, generalmente baja, una parte de los genes de los nuevos individuos puede ser modificado.

Su propósito es mantener diversidad dentro de la población y evitar un mínimo local prematuramente.

Utilizar solamente el operador de mutación es equivalente a recorrer aleatoriamente un espacio de búsqueda.

Mutación y Selección sin Cruce, es equivalente a un algoritmo tipo hill-climbing.

OTROS EFECTOS DE OPERADORES GENÉTICOS

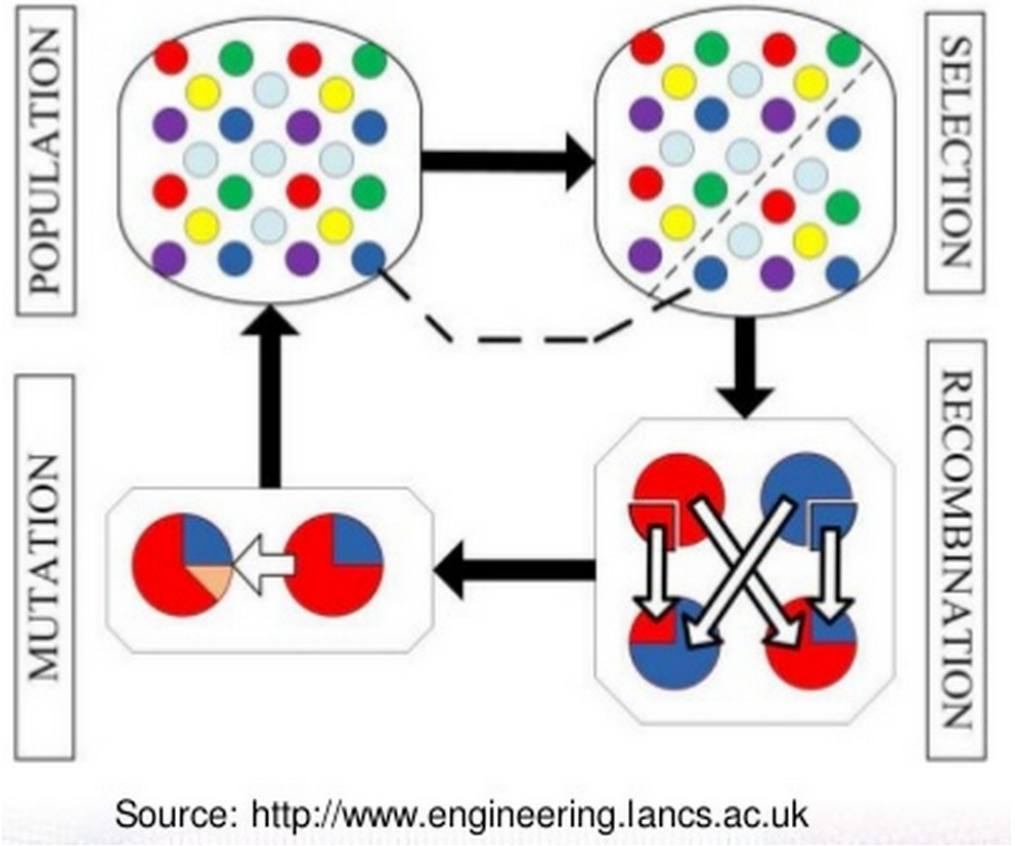
Utilizar solamente operador de selección tenderá a formar poblaciones que sean copias de los mejores individuos de la población de la generación anterior.

Utilizar solamente selección y cruce puede llevar a obtener soluciones sub-optimas pero de una región local del espacio de búqueda. Se pueden perder zonas del espacio de búsqueda que contengan mejores soluciones.

Algoritmos Genéticos - Metaheurística

1. Generar una población inicial.
2. Evaluar el fitness de cada individuo de la población.
3. Mientras la condición de fin no se cumpla, realizar:
    a) Seleccionar individuos para la reproducción.
    b) Cruzar los individuos seleccionados.
    c) Mutar nuevos individuos.
    d) Evaluar el fitness de los nuevos individuos.
    e) Generar la nueva población.

Algoritmos Genéticos (gráficamente)



Algoritmos genéticos (Ejemplo 1)

Cantidad de genes (N): 10

Cantidad de individuos (P): 6

Representación: cada gen será representado mediante el alfabeto {0, 1}.

Ejemplo de individuo: “1001101011”

Función objetivo:

$$maximizar \sum_{j=1}^{6} \sum_{i=1}^{10} gen_{i,j}$$

Algoritmos genéticos (Ejemplo 1)

Initial Population

$s_1 = 1111010101$

$s_2 = 0111000101$

$s_3 = 1110110101$

$s_4 = 0100010011$

$s_5 = 1110111101$

$s_6 = 0100110000$

Evaluate Initial Population

$s_1 = 1111010101 \quad f(s_1) = 7$

$s_2 = 0111000101 \quad f(s_2) = 5$

$s_3 = 1110110101 \quad f(s_3) = 7$

$s_4 = 0100010011 \quad f(s_4) = 4$

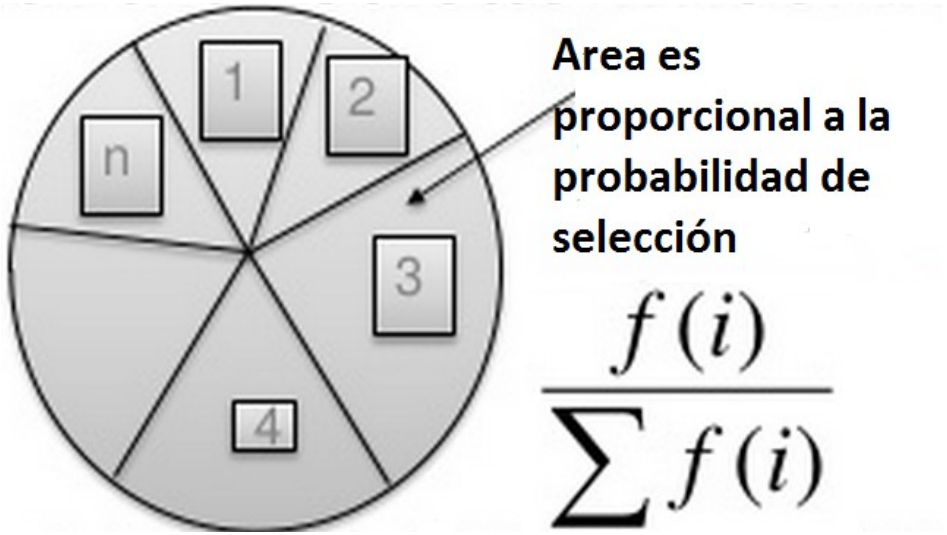
$s_5 = 1110111101 \quad f(s_5) = 8$

$s_6 = 0100110000 \quad f(s_6) = 3$

$= 34$

Algoritmos genéticos (Ejemplo 1)

Selección: Método de la ruleta



Se gira la ruleta P veces (la cantidad de individuos de la población). Una vez la ruleta se detiene, se selecciona al individuo que está bajo la aguja de la ruleta.

El área que corresponde a cada individuo en la ruleta es directamente proporcional a la probabilidad de selección asociada a dicho individuo, la cual es obtenida mediante la fórmula mostrada en la figura anterior.

Algoritmos genéticos (Ejemplo 1)

Método de la ruleta sesgada:

1. Obtener para cromosoma de la población su puntuación o fitness $f(x_i)$.

2. Calcular para cada cromosoma la suma acumulada de fitness, esto es sumar su fitness más el fitness de cada uno de los cromosomas que le preceden.

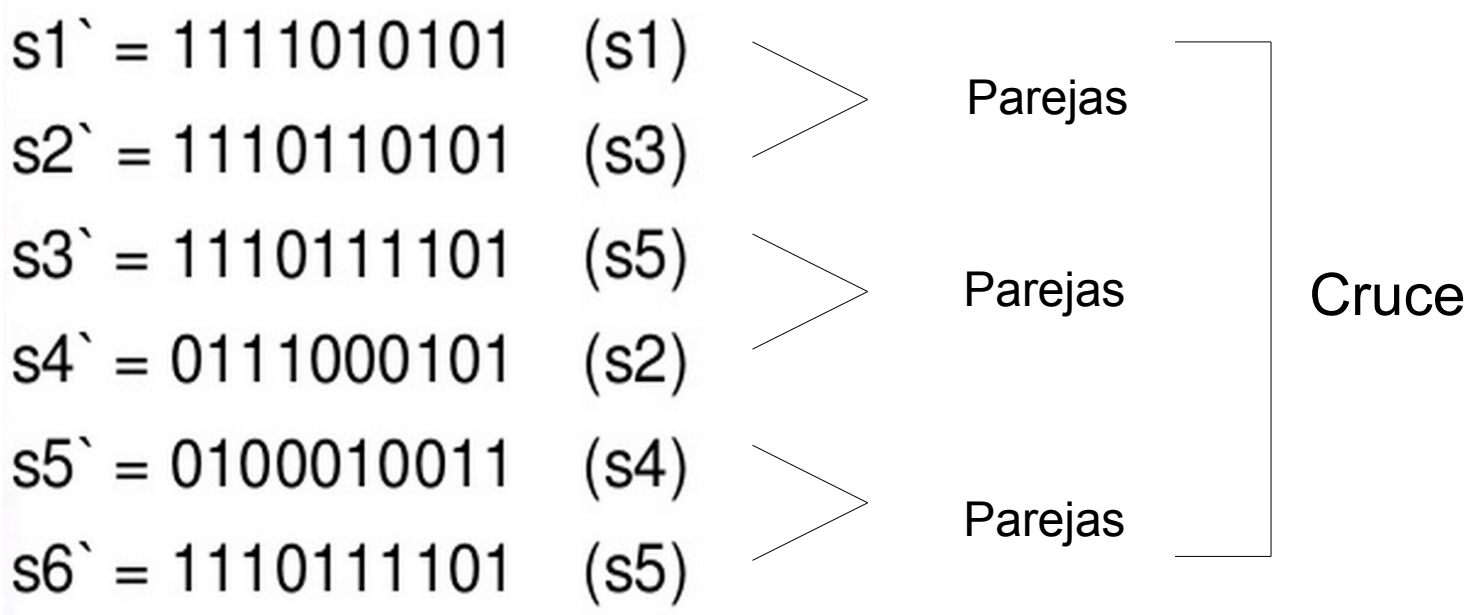
3. Elegir un número aleatorio entre 0 y la suma total de fitness (ambos inclusive).

4. El primer cromosoma cuyo fitness acumulado (paso 2) sea mayor o igual al número aleatorio obtenido, será el cromosoma seleccionado.

Se repetirá los pasos 3 y 4 hasta obtener P individuos.

Algoritmos genéticos (Ejemplo 1)

Posible resultado de la selección



Algoritmos genéticos (Ejemplo 1)

Por cada pareja el cruce se realiza con una probabilidad P_{cruce}(Ejemplo: 0.6).

Supongamos que son seleccionadas las parejas 1 y 3.

Por cada pareja, se selecciona aleatoriamente un punto de cruce (Ejemplo: para la pareja 1, el punto de cruce 2, y para la pareja 3, el punto de cruce es 5).

Antes del cruce: Pareja 1

S1' → 11**11010101**
S2' → 11**10110101**

Antes del cruce: Pareja 3

S5' → 01000**10011**
S6' → 11101**11101**

Después del cruce: Descendientes

S1" → 11**10110101**
S2" → 11**11010101**

S5" → 01000**11101**
S6" → 11101**10011**

Los genes de los cromosomas S3 y S4 se copian a los cromosomas S3" y S4", respecitvamente.

Algoritmos genéticos (Ejemplo 1)

La mutación se aplica a cada gen de cada cromosoma con una probabilidad $P_{mutacion.}$
Generalmente esta probabilidad es muy baja (Ejemplo: 0.1).

S1" → 1110110101
S2" → 1111010101
S3" → 1110111101
S4" → 0111000101
S5" → 0100011101
S6" → 1110110011

Mutación

S1''' → 1110100101
S2''' → 1111110100
S3''' → 1110101111
S4''' → 0111000101
S5''' → 0100011101
S6''' → 1110110001

Algoritmos genéticos (Ejemplo 1)

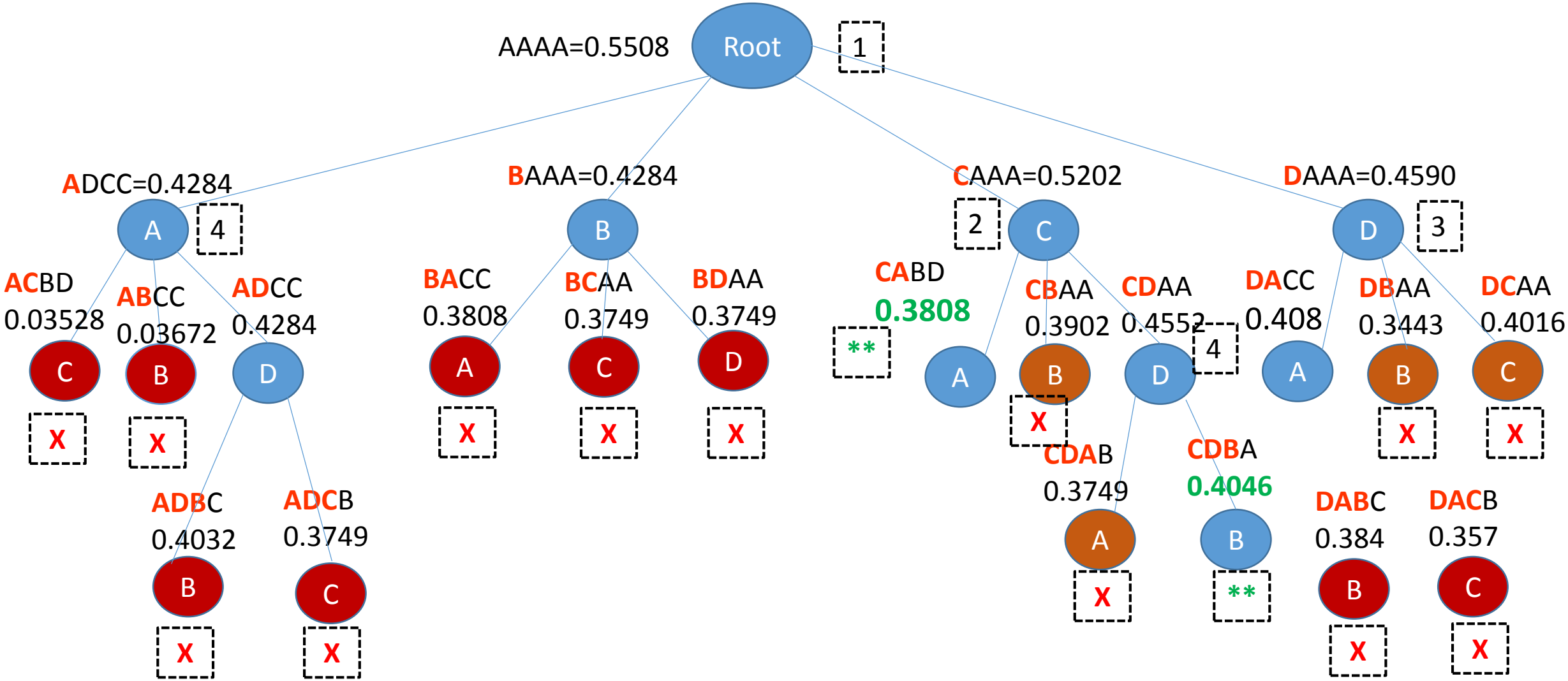
S1''' → 1110100101 → fitness = 6
S2''' → 1111110100 → fitness = 7
S3''' → 1110101111 → fitness = 8
S4''' → 0111000101 → fitness = 5
S5''' → 0100011101 → fitness = 5
S6''' → 1110110001 → fitness = 6

Función objetivo: 37

Optimización Exacta – Ejemplo – Branch and Bound

Una empresa tiene un equipo de trabajadores que debe llevar a cabo determinadas operaciones. Los miembros de ese equipo son los trabajadores A, B, C, y D. Las operaciones que deberán ser realizadas son O1, O2, O3, y O4. Cada miembro del equipo deberá realizar una única operación y todas las operaciones deberán ser realizadas satisfactoriamente. Sin embargo, se conoce por cada par formado, por un trabajador y una operación, la probabilidad de éxito de dicho trabajador en dicha operación. La tabla que se muestra a continuación indica dichas probabilidades.

		Operaciones			
		O1	O2	O3	O4
Miembros del Equipo	A	0.9	0.8	0.9	0.85
	B	0.7	0.6	0.8	0.7
	C	0.85	0.7	0.85	0.8
	D	0.75	0.7	0.75	0.7


AAAA=0.5508
Root
1
ADCC=0.4284
A
4
BAAA=0.4284
B
CAAA=0.5202
2
C
DAAA=0.4590
D
3
ACBD
0.03528
ABCC
0.03672
ADCC
0.4284
BACC
0.3808
BCAA
0.3749
BDAA
0.3749
CABD
0.3808
**
CBAA
0.3902
CDAA
0.4552
4
DACC
0.408
DBAA
0.3443
DCAA
0.4016
ADBC
0.4032
ADCB
0.3749
CDAB
0.3749
CDBA
0.4046
DABC
0.384
DACB
0.357
X
**

Problema de cobertura de un conjunto – Formulación Formal

Consideramos que existe un determinado conjunto finito $M, M_j \subseteq M, j \in \{1, \dots, n\}$ son n sub-conjuntos de M, y cada sub-conjunto tiene un coste/peso asociado $c_j, j \in \{1, \dots, n\}$.

Cobertura del conjunto

Considerar:

- Una colección $T \subseteq \{1, \dots, n\}$ tal que $\bigcup_{j \in T} M_j = M$.
- A es una matriz de incidencia y e es un vector de números 1.

Luego, la siguiente expression define formalmente el problema:

$$\min_{T} \left\{ \sum_{j \in T} c_j : T \text{ is a cover} \right\} = min \left\{ \sum_{j=1}^{n} c_j x_j : Ax \geq e, x \in \{0,1\}^n \right\}$$

Problema de cobertura de un conjunto – Ejemplo

Un hospital necesita mantener una plantilla de médicos de guardia, tal que si es necesario realizar un procedimiento, exista por lo menos un médico disponible para llevarlo a cabo.

En el hospital existe una lista de procedimientos que pueden ser requeridos cada día, y por cada médico que es llamado de urgencia para llevar a cabo alguno de estos procedimientos se paga una tarifa determinada.

Se necesita obtener un conjunto de médicos mediante el cual se pueda cubrir todos los procedimientos del hospital a un mínimo coste. Considere que no es necesario llevar a cabo los procedimientos en paralelo.

Problema de cobertura de un conjunto – Ejemplo (cont.)

	Médico1	Médico2	Médico3	Médico4	Médico5	Médico6
Procedimiento1	**SI**			**SI**		
Procedimiento2	**SI**				**SI**	
Procedimiento3		**SI**	**SI**			
Procedimiento4	**SI**					**SI**
Procedimiento5		**SI**	**SI**			**SI**
Procedimiento6		**SI**				

Representación de datos

Tenemos m=6 procedimientos que pueden ocurrir en una noche de guardia y n=6 médicos disponibles. Si en un determinado momento se requiere atención para un procedimiento debe existir por lo menos un médico que se pueda encargar del mismo.

Representación de datos

Que un médico de guardia j esté disponible cuesta al hospital c_j.

Variables: $x_j = 1$, si el médico j forma parte del conjunto solución, es decir si estará disponible para una llamada de urgencia. Sino, su valor es 0.

Modelo formulado mediante programación matemática.

$$\min \sum_{j=1}^{n} c_j x_j$$

Importe total que deberá pagar el hospital.

s.t.

$$\sum_{j=1}^{n} a_{ij} x_j \geq 1, \forall i, i \in \{1, \dots, m\}$$

Cuando un procedimiento deba ser realizado, por lo menos un médico calificado para el mismo debe estar en plantilla de urgencia.

$$x_j \in \{0,1\}, \forall j, j \in \{1, \dots, n\}$$

x_j es una variable binaria.

Problema de cobertura de un conjunto
Modelo para GLPKSOL

```
/* número de procedimientos requeridos */
param m, integer, > 0;

/* número de médicos, disponibles para ser
seleccionados como parte de la plantilla de guardia */

param n, integer, > 0;

/* conjunto de índices para hacer referencia a cada
procedimiento */
set I := 1..m;

/* conjunto de índices para hacer referencia a cada
médico */
set J := 1..n;
```

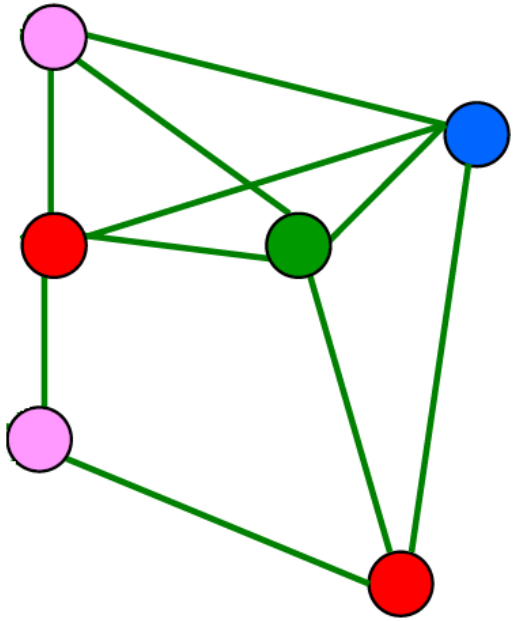
Modelo para GLPKSOL (cont.)

```
/* coste de tener al médico j en la plantilla de urgencia */
param c{j in J}, >= 0;

/* a[i,j]=1 indica que el médico j está calificado
para realizar el procedimiento i.*/

param a{i in I, j in J}, binary;

/* x[j] = 1 significa que el médico j formará parte de
la plantilla de urgenica. Sino, su valor será 0. */

var x{j in J}, binary;

/* cuando el procedimiento i es requerido, debe por lo menos
estar en plantilla un médico que sea capaz de llevarlo a cabo. */
s.t. cover{i in I}: sum{j in J} a[i,j]*x[j] >= 1;
```

```
/* se busca minimizar el importe total que deberá pagar el hospital por la plantilla
seleccionada */
minimize obj: sum{j in J} c[j] * x[j];
solve;
printf{j in J} "%d\n", x[j];

data;
param m := 6;
param n := 6;
param c := 1 10, 2 15, 3 2, 4 1, 5 5, 6 8;
param a : 1  2  3  4  5  6 :=
    1   1  0  0  1  0  0
          2   1  1  0  0  1  0
          3   0  1  1  0  0  0
          4   1  0  0  0  0  1
          5   0  0  1  0  0  0
          6   0  0  0  0  0  1  ;
end;
```

Problema: Asignar un color a cada vértice

Se considera un grafo formado por un conjunto de vértices y aristas. El problema consiste en asignar un color a cada vértice tal que dos nodos adyacentes no tengan asignado el mismo color. El objetivo es resolver el problema utilizando la mínima cantidad de colores.



Ejemplo: Asignación de frecuencias. Dos estaciones base no pueden compartir la misma frecuencia por la interferencia que eso ocasionaría.

Modelar el problema

Conjunto de colores: $C = \{1, \dots, |V|\}$,
Conjunto de vertices: V
Conjunto de aristas: E

$$y_k = \begin{cases} 1 & \text{Si el color k es usado para colorear algún vértice} \\ 0 & \text{Color k no es utilizado} \end{cases}$$

$$x_{ik} = \begin{cases} 1 & \text{Si el nodo i es pintado con el color k} \\ 0 & \text{En otro caso} \end{cases}$$

Formalización del problema

$$\min \sum_{k \in C} y_k$$

s.t. $\sum_{k \in C} x_{ik} = 1, \forall i \in V$

$x_{ik} \leq y_k, \forall i \in V, \forall k \in C$

$x_{ik} + x_{jk} \leq 1, \forall (i,j) \in E, \forall k \in C$

$x_{ik} \in \{0,1\}, \forall i \in V, \forall k \in C$

$y_k \in \{0,1\}, \quad \forall k \in C$

Modelar el problema para resolverlo en GLPKSOL

```
/* Problema de colorear el grafo*/
/* Escrito en GNU MathProg por Andrew Makhorin <mao@gnu.org> */

/* número de vértices */
param n, integer, >= 2;
param nc, integer, >0;
/* conjunto de índices para hacer referencia a cada vértice */
set V := {1..n};

/* conjunto de arcos */
set E, within V cross V;

/* no puede haber ciclos */
check{(i,j) in E}: i != j;
```

Colorear vertices del grafo
Modelar el problema para resolverlo en GLPKSOL (cont.)

```
/* x[i,c] = 1 significa que el color c es asignado al nodo i */
var x{i in V, c in 1..nc}, binary;

/* u[c] = 1 significa que el color c es usado por algún nodo */
var u{c in 1..nc}, binary;

/* a cada nodo se le debe asignar exactamente un color */
s.t. map{i in V}: sum{c in 1..nc} x[i,c] = 1;

/* no se puede asignar un mismo color a nodos adyacentes */
s.t. arc{(i,j) in E, c in 1..nc}: x[i,c] + x[j,c] <= u[c];

/* el objetivo es minimizer el número de colores utilizados */
minimize obj: sum{c in 1..nc} u[c];
```

Colorear vertices del grafo
Modelar el problema para resolverlo en GLPKSOL (cont.)

```
/* Estos datos corresponden a la instancia myciel3.col de:
http://mat.gsia.cmu.edu/COLOR/instances.html */

data;
param n := 11;
param nc := 11;
set E := 1 2
         1 4
         1 7
         1 9
         2 3
         2 6
         2 8
         3 5
```

Colorear vértices del grafo
Modelar el problema para resolverlo en GLPKSOL (cont.)

```
3 7
3 10
4 5
4 6
4 10
5 8
5 9
6 11
7 11
8 11
9 11
10 11;

end;
```

Tabú Search

El algoritmo Tabu Search fue presentado por primera vez en 1986 por Fred W. Glover.

Se trata de un procedimiento iterativo que realiza una exploración guiada en el espacio de soluciones a tratar. Aunque no se conoce una demostración formal que explique su buen comportamiento, empíricamente se puede observar el exitoso desempeño en los problemas de optimización combinatoria, donde los métodos de optimización exacta no pueden encontrar soluciones en un tiempo de cómputo aceptable.

Algunas de las principales características que describen a esta meta-heurística son:

- El algoritmo parte de una solución inicial, que se va modificando y obteniendo diferentes soluciones
- Acepta peores soluciones que la mejor encontrada hasta el momento
- Utiliza una lista “tabú”, cuyo objetivo es forzar al algoritmo a explorar nuevas soluciones, y evitar la repetición de soluciones
- Puede utilizar una función de aspiración, la cual permite elegir uno o varios elementos prohibidos, según la lista tabú de una solución en particular.

Tabú Search - Características

Solución Inicial

El primer paso de este algoritmo es obtener una solución inicial. Esto es una asignación factible de las variables del problema que cumpla con todas las restricciones del mismo. Se puede obtener mediante la asignación aleatoria de las variables hasta encontrar una solución factible, o mediante la asignación consecutiva de las variables de tal forma que se vayan cumpliendo las restricciones, o bien, mediante alguna heurística según el dominio del problema que se intente resolver.

Función Objetivo

Se trata de una función cuyo dominio son las variables de decisión y cuyo valor indica la calidad de cada solución encontrada. El objetivo de la optimización puede ser minimizar o maximizar dicha función.

Tabú Search - Características

Vecindad

Se entiende, desde una solución en particular. Es el espacio de soluciones que se puede alcanzar desde una determinada solución, a partir de modificaciones realizadas en algunas de sus variables de decisión. Estas modificaciones se deben llevar a cabo de acuerdo a un procedimiento determinado, que es el mismo durante todo el proceso de optimización.

Mínimo Local

También llamado mínimo relativo, es un mínimo que también puede ser un mínimo global. Formalmente diremos que una función $f(x)$ alcanza un mínimo local en un punto a de su dominio si existe un entorno de a en el que los valores de f son mayores que $f(a)$. Si la función es derivable en a, entonces $f'(a) = 0$.

Tabú Search - Características

Movimiento

Un movimiento es una variación a la solución actual, que nos permite cambiar de una vecindad a otra, es decir, un espacio de soluciones a otro.

Lista Tabú

La lista tabú sirve para prohibir soluciones o parte de las mismas durante un número de iteraciones. Su objetivo es evitar que el algoritmo repita soluciones. Lo que busca el algoritmo de Tabú Search, es escapar de un mínimo local permitiéndole desplazarse de esta forma hacia una nueva vecindad.

Función de Aspiración

Hay veces que necesitamos relajar las decisiones cuando una función tabú resulta atractiva. En ese caso la función de aspiración nos permite realizar movimientos prohibidos que se encuentren en la lista tabú, bajo un criterio o umbral establecido; por ejemplo cuando estos permiten alcanzar una mejor solución respecto de la mejor encontrada hasta el momento.

Tabú Search - Características

Criterio de parada

Se trata de la condición que debe cumplirse para que el proceso de optimización se detenga. Puede ser una cantidad determinada de iteraciones, o una cantidad determinada de iteraciones donde la mejor solución no mejore. Otras condiciones pueden ser: alcanzar un determinado tiempo de ejecución, o que la función objetivo de una solución alcance un determinado valor (establecer un umbral).

Intensificación

La intensificación es una estrategia de búsqueda en el marco de tabú search, que tiene por objeto explorar en mayor detalle una determinada zona del espacio de soluciones o vecindad, con la expectativa de encontrar una solución mejor a las conocidas. La intensificación puede estar motivada por alguna característica sobresaliente de una solución o conjunto de soluciones, sea en su calidad o basada en algún conjunto de atributos que estén asociados a buenas soluciones.

Tabú Search - Características

Existen diversas maneras de implementar esta técnica, la más simple consiste en disminuir el tamaño de la lista tabú, de modo que se permitan más movimientos de retroceso que, a riesgo de generar la ocurrencia de ciclos, permitan una exploración más detallada de una determinada zona. Otra alternativa consiste en modificar el criterio de selección de movimientos a fin de privilegiar aquellos que estén asociados a atributos que aparecen frecuentemente en soluciones buenas.

Diversificación

Contraria a la estrategia de intensificación, la diversificación pretende analizar regiones o vecindades lejanas de la actual, en el espacio global de soluciones, con la intención de escapar a mínimos locales y evitar que queden vecindades sin visitar. Una forma simple de implementar este comportamiento consiste en aumentar el tamaño de la lista tabú. De esta forma, al aumentar la cantidad de soluciones prohibidas en una zona dada, se fuerza al algoritmo a orientar la búsqueda a otras áreas lejanas a la actual.

Tabú Search - Características

Otra opción es modificar las reglas de selección de movimientos, a fin de privilegiar la utilización de atributos que no fueron empleados frecuentemente en las soluciones analizadas hasta el momento.

Tamaño de la lista tabú

La variación dinámica del tamaño de la lista tabú, permite almacenar una cantidad mayor o menor de movimientos o soluciones prohibidas. Un aumento en el tamaño de la lista tabú puede verse como una diversificación, mientras que una disminución puede considerarse como una intensificación. La variación en el tamaño de la lista podría estar motivada en base a la frecuencia de aparición de determinados atributos en las soluciones que se visitaron recientemente, o bien en un componente aleatorio. La combinación de intensificación y diversificación permite una exploración amplia y detallada del espacio de soluciones aumentando las posibilidades de explorar nuevas vecindades para luego intentar encontrar mejores mínimos locales.

Tabú Search - Características

Soluciones Elite

Se llaman soluciones élite a aquellas buenas soluciones que fueron exploradas durante el proceso de búsqueda y que por algún motivo se destacan del resto, ya sea por alguna característica en sus atributos o por tener valores de la función objetivo cercanos al mejor valor conocido. En general estas soluciones se las almacena en una lista para ser exploradas mediante un proceso de intensificación en momentos posteriores de la búsqueda.

Frecuencia

La idea de este punto es identificar atributos, movimientos o soluciones completas que se repiten en las soluciones buenas. El objetivo es establecer un patrón que ayude a tomar una decisión.

Tabú Search – Meta-heurística

1. Determinar una solución inicial $i \in V$, donde V conforma el conjunto global de soluciones.

2. Generar un subconjunto V^* de soluciones tal que $V^* \subseteq V$ que no sean Tabú o que cumplan con el criterio de Aspiración.

3. Encontrar la mejor solución $j \in V^*$, respecto a la función objetivo $f \,/\, f(j) \leq f(k)\ \forall k \in V^*$.

4. Hacer $i = j$.

5. Actualizar la lista tabú y la función de Aspiración.

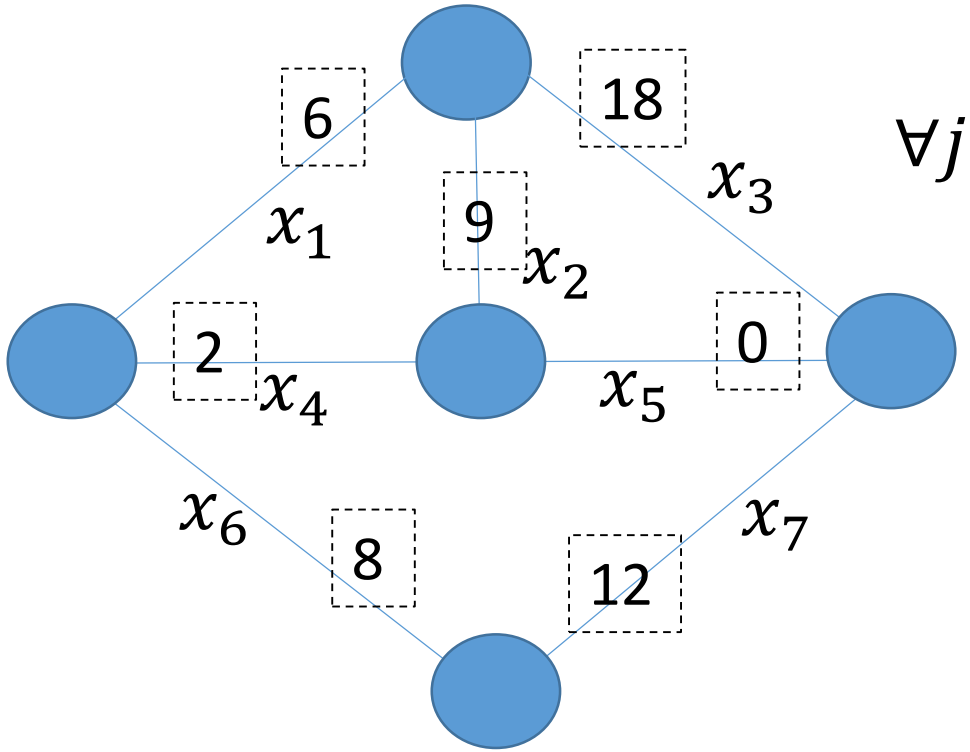
6. Si cumple con el criterio de parada, entonces finalizar el procedimiento. Sino, volver al paso 2.

Tabú Search – Ejemplo

Problema

Encontrar un spanning tree con mínimo coste considerando ciertas restricciones que deben cumplirse sobre las aristas que formen el árbol de la solución.

Instancia del problema



Variables de decisión

$$\forall j \in \{1, \dots, 7\} \begin{cases} x_j = 1 & \text{Arista } j \text{ es parte del spanning tree.} \\ x_j = 0 & \text{Arista } j \text{ no es parte del spanning tree.} \end{cases}$$

Restricciones

$$subject\ to \quad x_1 + x_2 + x_6 \leq 1$$
$$x_1 \leq x_3$$

$penalización\ por\ incumplimiento\ de\ restricción = 50$

$$objetivo: \min \sum_{i=1}^{7} x_i c_i$$

Lista Tabú: Se actualizará en cada iteración por agregar la arista que se ha añadido al spanning tree en la misma iteración.

Movimiento Tabú: no se podrá eliminar del grafo una arista que sea una de las dos últimas aristas agregadas a la lista tabú.

Criterio de aspiración: se puede realizar un movimiento tabú si dicho movimiento genera una solución mejor que la obtenida hasta entonces.

Observación: Se aceptan movimientos que generen soluciones no factibles, aunque su coste conlleva la penalización correspondiente.

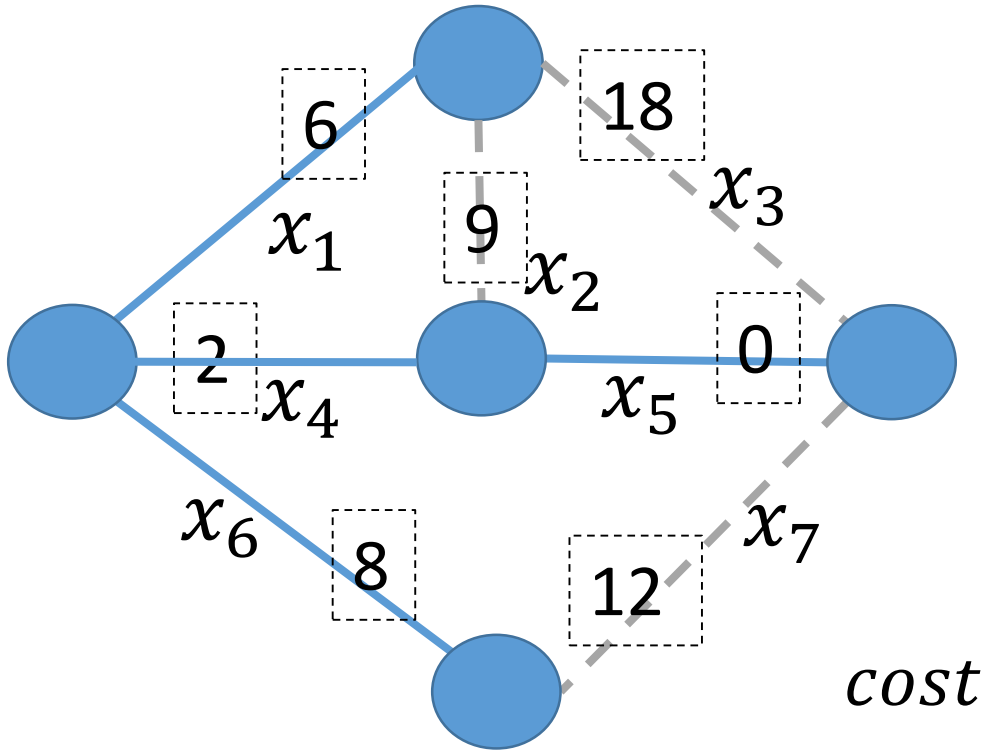
Restricciones

$$subject\ to\quad x_1 + x_2 + x_6 \leq 1$$
$$x_1 \leq x_3$$

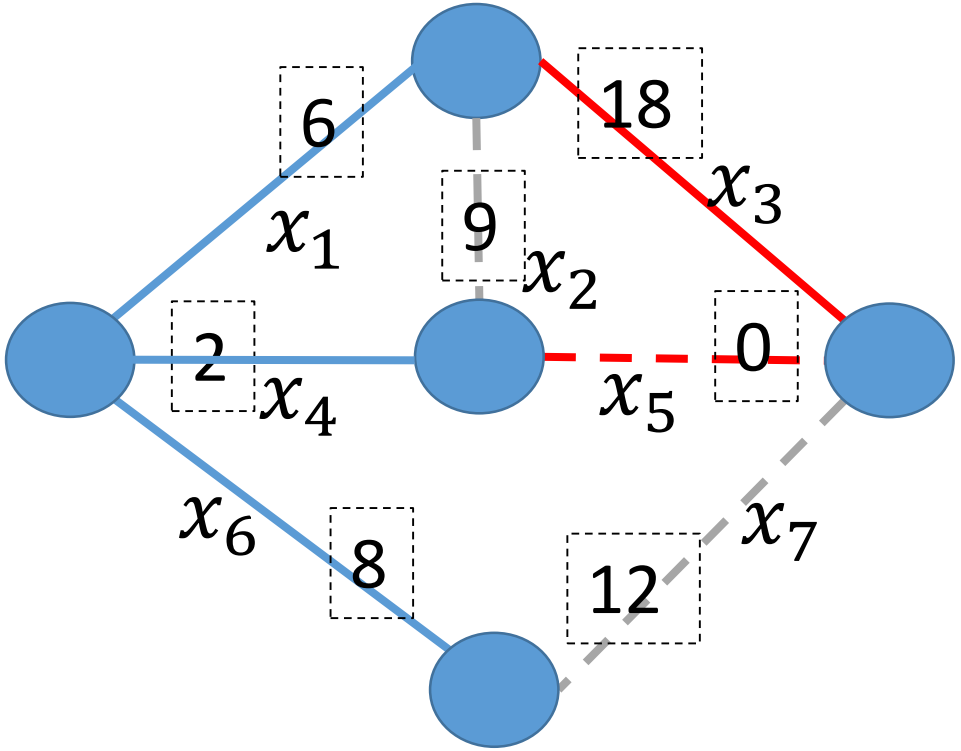
$penalización\ por\ incumplimiento\ de\ restricción = 50$

$$objetivo: \min \sum_{i=1}^{7} x_i c_i$$

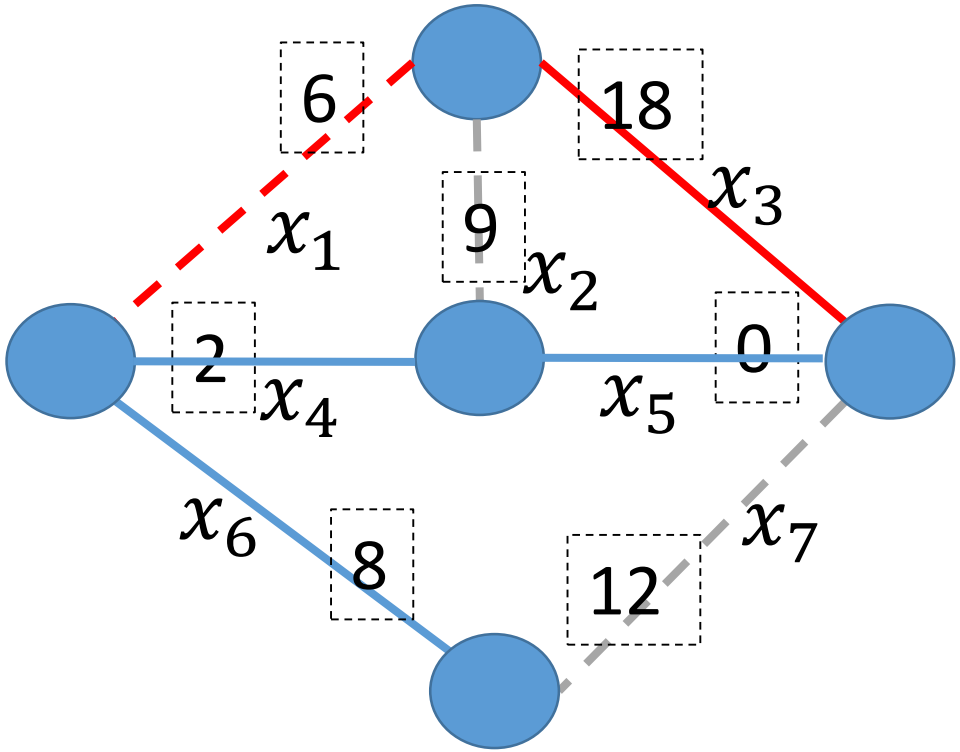
Solución Inicial



$$cost = 6x_1 + 2x_4 + 0x_5 + 8x_6 + 50 + 50 = 116$$

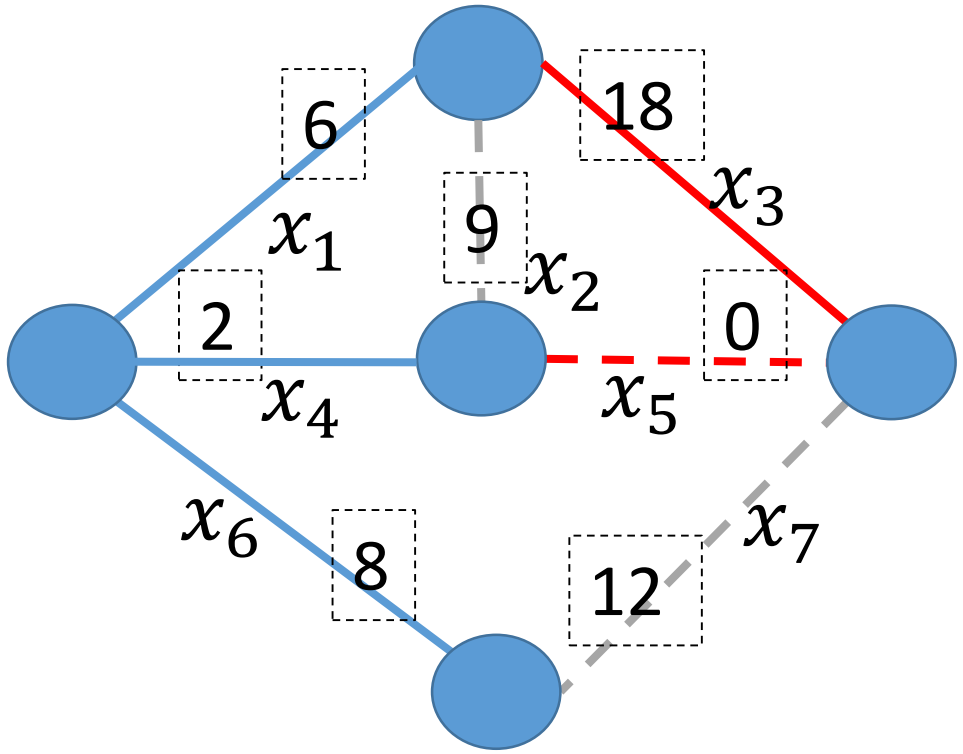


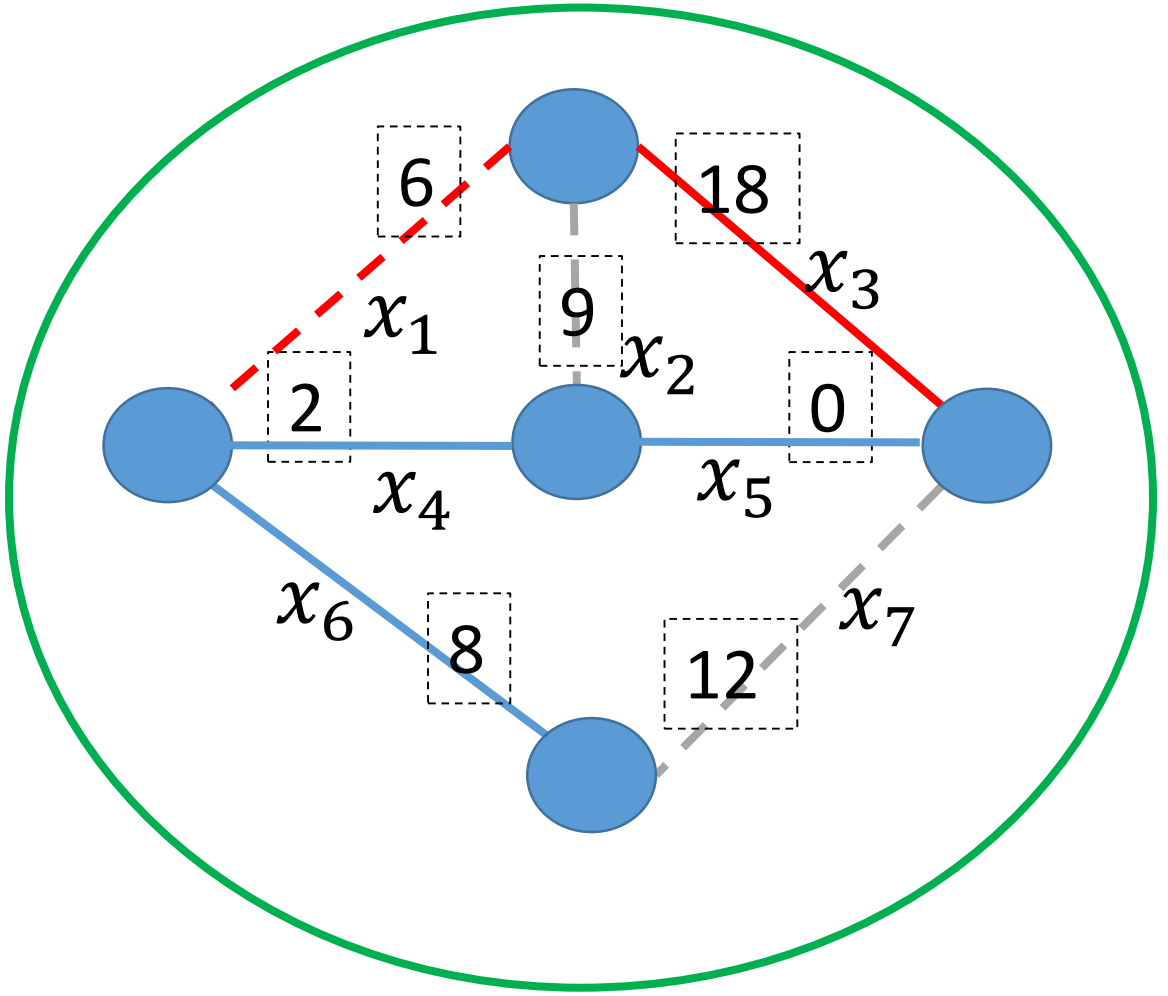
-X5 ; +X3
cost = $6x_1 + 18x_3 + 2x_4 + 8x_6 + 50 = 84$



-X1 ; +X3
cost = $18x_3 + 2x_4 + 0x_5 + 8x_6 = 28$

$$subject\ to \quad x_1 + x_2 + x_6 \leq 1$$
$$x_1 \leq x_3$$



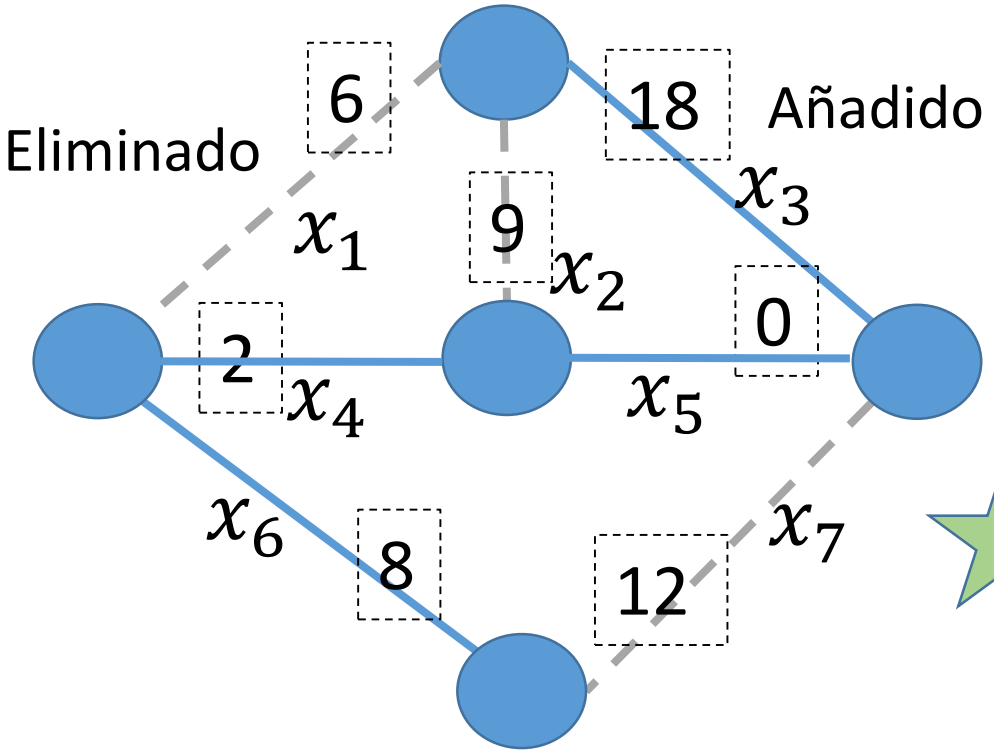


$$subject\ to \quad x_1 + x_2 + x_6 \leq 1$$
$$x_1 \leq x_3$$

-X1; +X3

cost = $18x_3 + 2x_4 + 0x_5 + 8x_6 = 28$

Iteración 2



Lista Tabú
x_3

Mejor solución encontrada hasta ahora

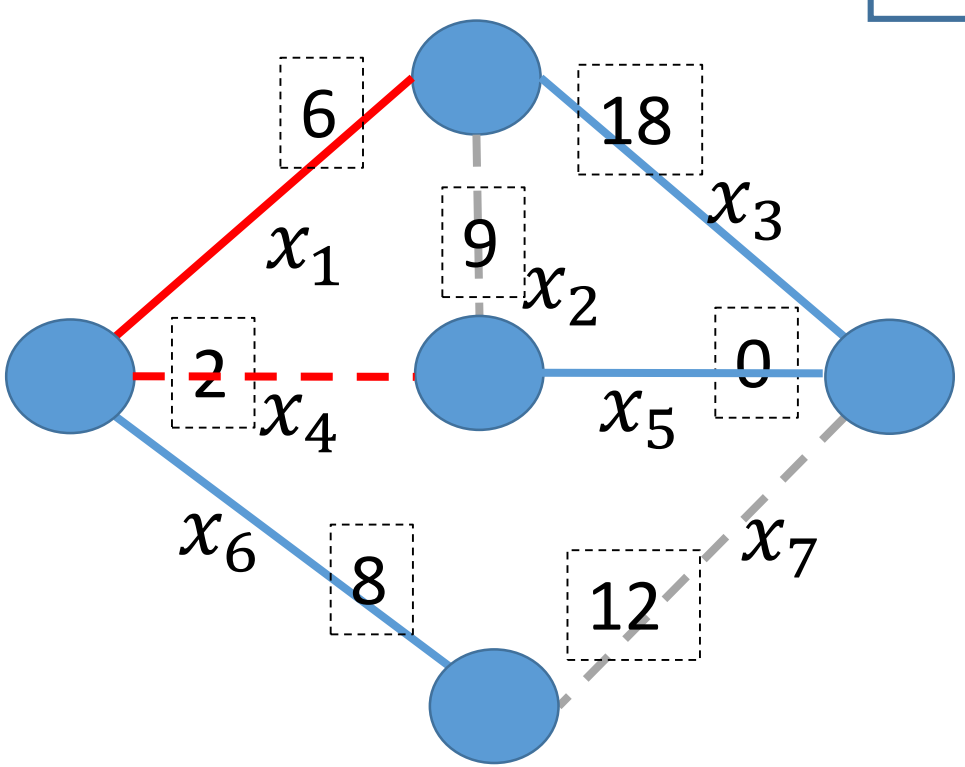
$$cost = 18x_3 + 2x_4 + 0x_5 + 8x_6 = 28$$

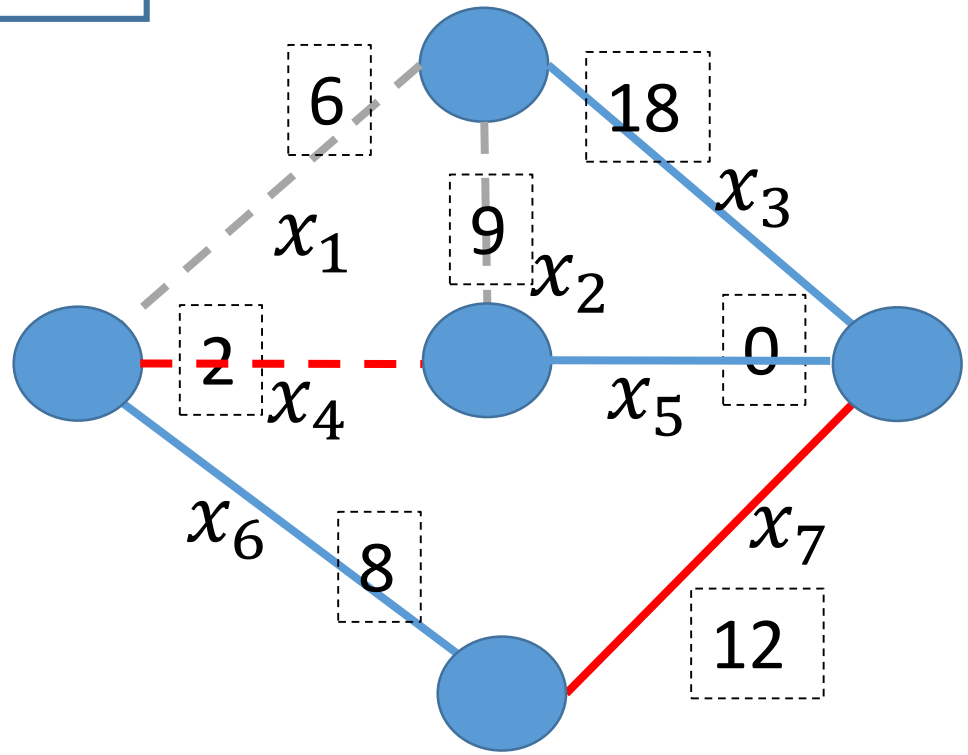
$$subject\ to \quad x_1 + x_2 + x_6 \leq 1$$

$$x_1 \leq x_3$$

Iteración 2

Lista Tabú
x_3



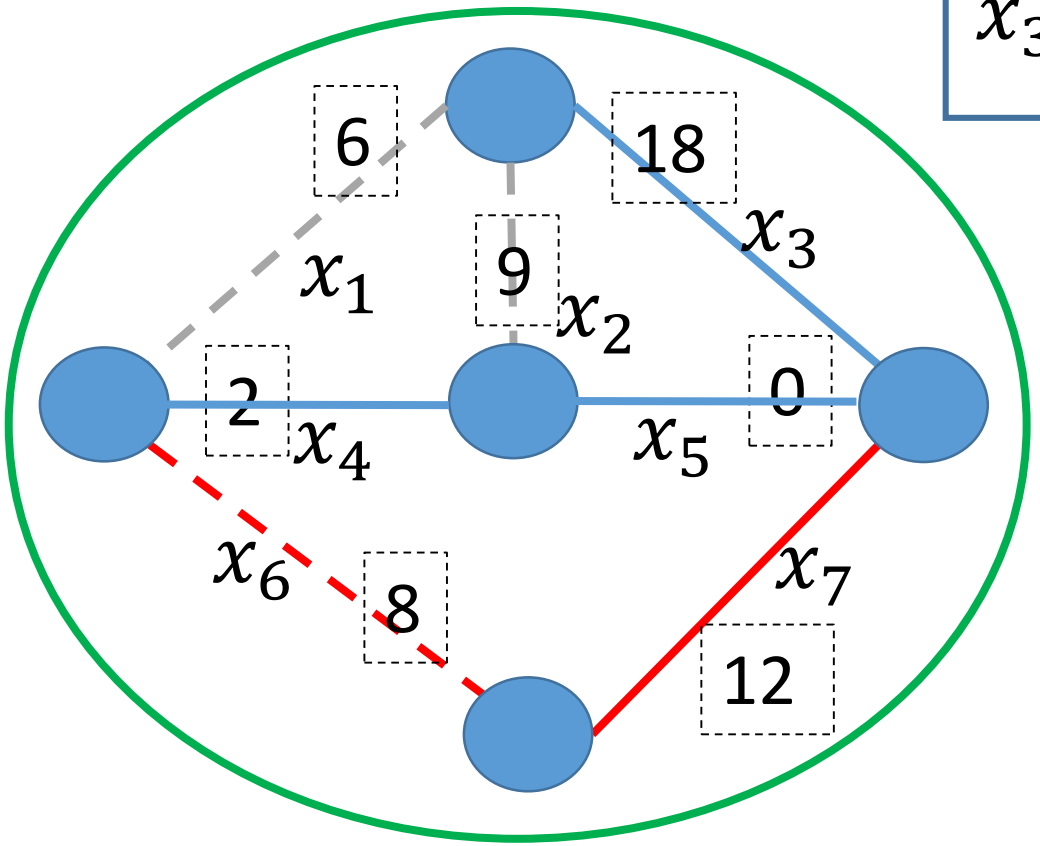


-X_4 ; + X_1
cost = $6x_1 + 18x_3 + 0x_5 + 8x_6 + 50 = 82$

-X_4 ; + X_7
cost = $18x_3 + 0x_5 + 8x_6 + 12x_7 = 38$

$$subject\ to \quad x_1 + x_2 + x_6 \leq 1$$
$$x_1 \leq x_3$$

Lista Tabú
x_3



-X_6 ; + X_7

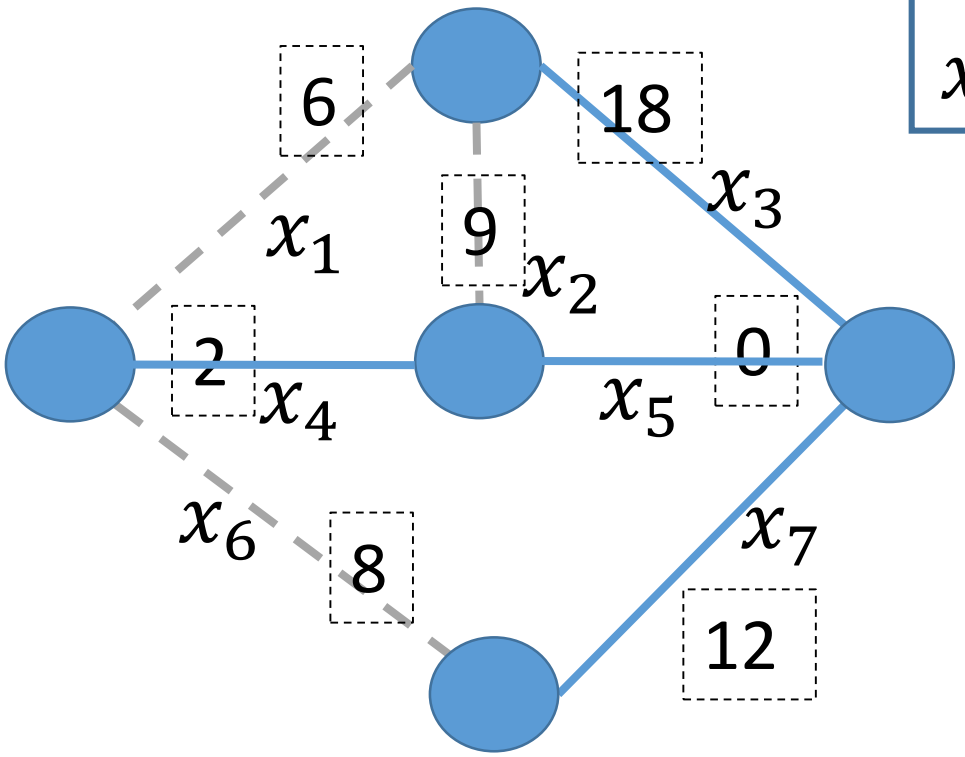
cost = $18x_3 + 2x_4 + 0x_5 + 12x_7 = 32$

$$subject\ to \quad x_1 + x_2 + x_6 \leq 1$$

$$x_1 \leq x_3$$

Iteración 3

Lista Tabú
x_3
x_7



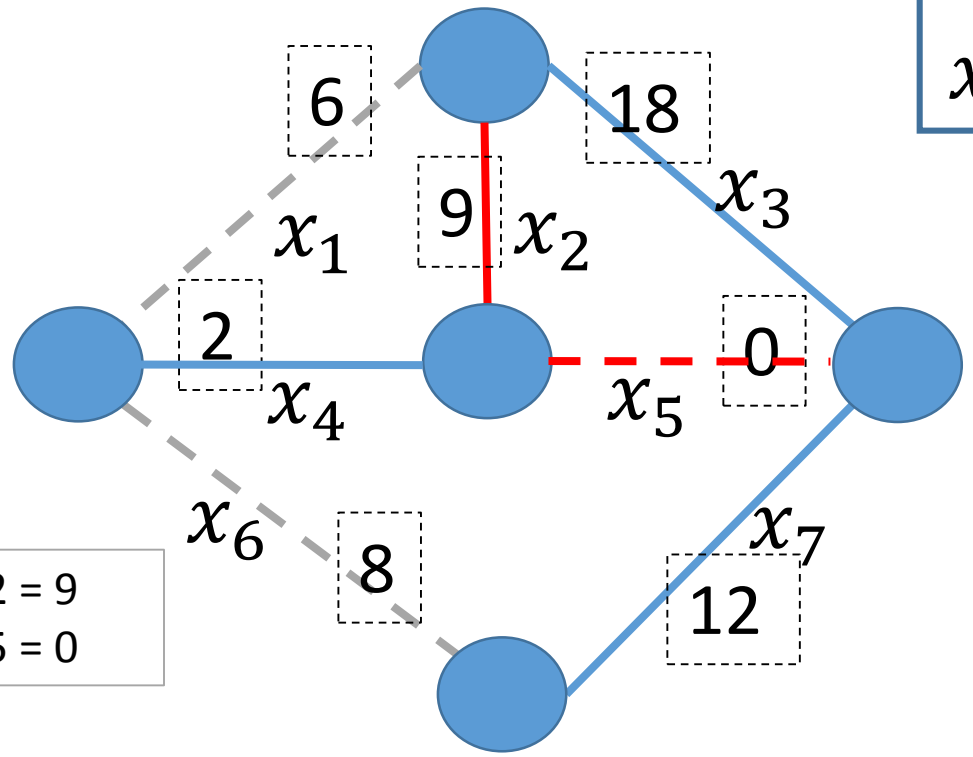
$$subject\ to \quad x_1 + x_2 + x_6 \leq 1$$

$$x_1 \leq x_3$$

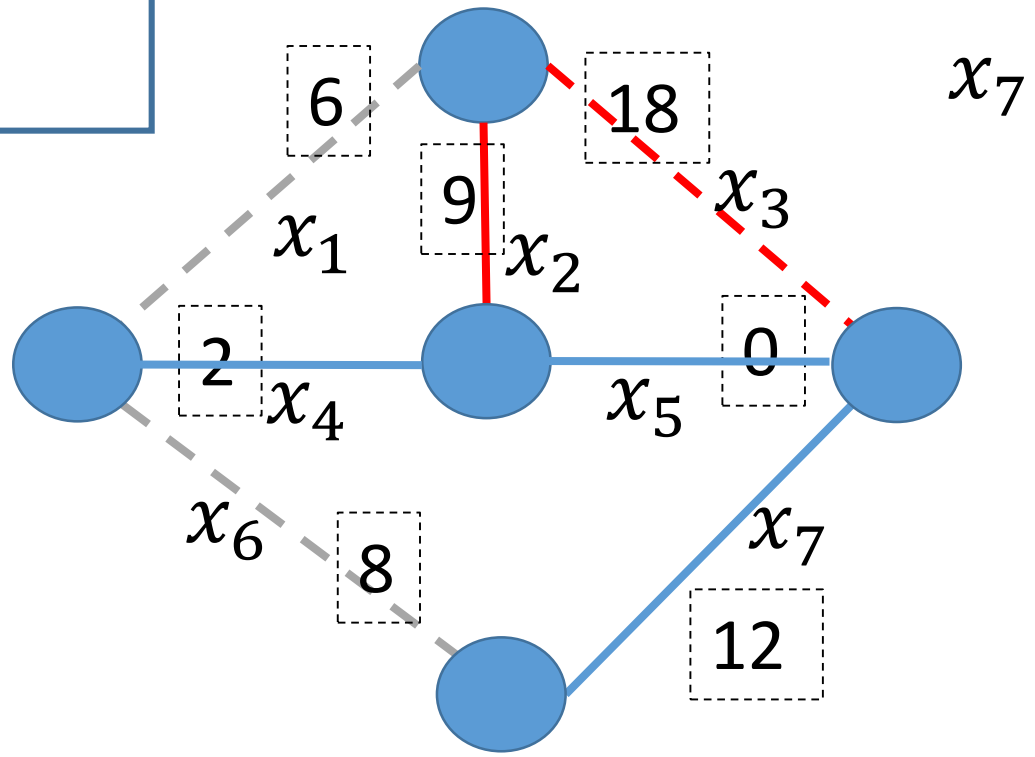
$$cost = 18x_3 + 2x_4 + 0x_5 + 12x_7 = 32$$

Iteración 3

Lista Tabú
x_3
x_7



X2 = 9
X5 = 0



-X_5 ; + X_2
cost = $9x_2 + 18x_3 + 2x_4 + 12x_7 = 41$

-X_3 ; + X_2
cost = $9x_2 + 2x_4 + 0x_5 + 12x_7 = 23$

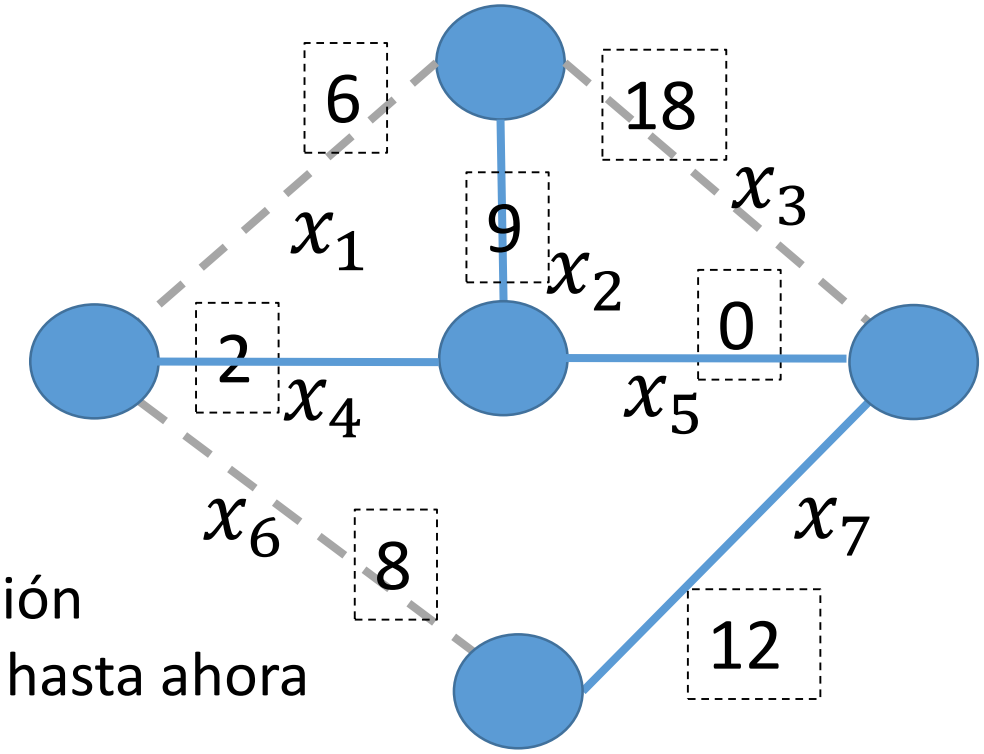
$$subject\ to \quad x_1 + x_2 + x_6 \leq 1$$
$$x_1 \leq x_3$$

$$cost = 9x_2 + 2x_4 + 0x_5 + 12x_7 = 23$$

Iteración 4

Lista Tabú
~~x_3~~
x_7
x_2

Criterio de aspiración aplicado



Mejor solución encontrada hasta ahora

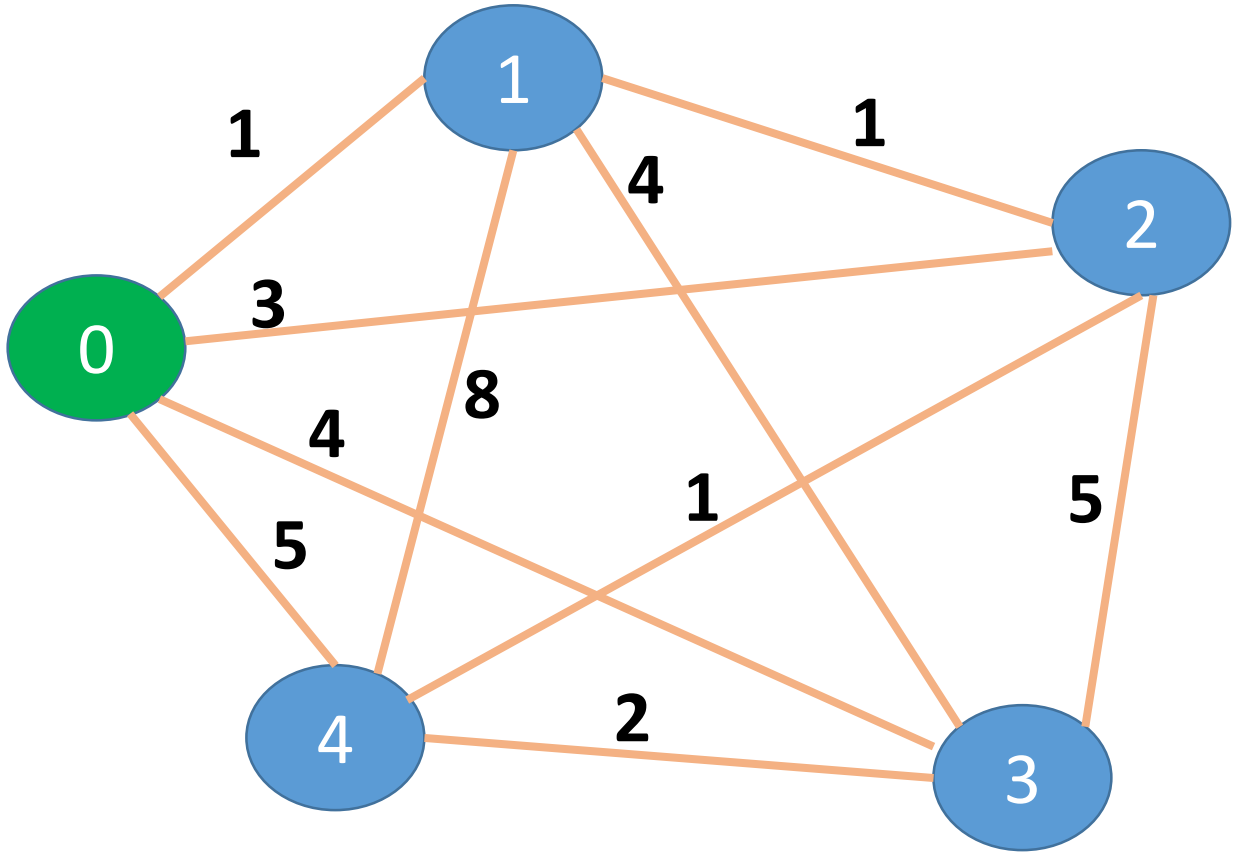
$$subject\ to \quad x_1 + x_2 + x_6 \leq 1$$
$$x_1 \leq x_3$$

$$cost = 9x_2 + 2x_4 + 0x_5 + 12x_7 = 23$$

Problema 2: Travelling Salesman Problem

Dado un conjunto finito de ciudades, un conjunto finito de rutas con sus correspondientes distancias (desde una ciudad a otra), indicar cuál es el camino más corto para salir de la ciudad 0 y volver a ella pasando por todas las ciudades una sola vez.

Instancia del problema



Lista Tabú

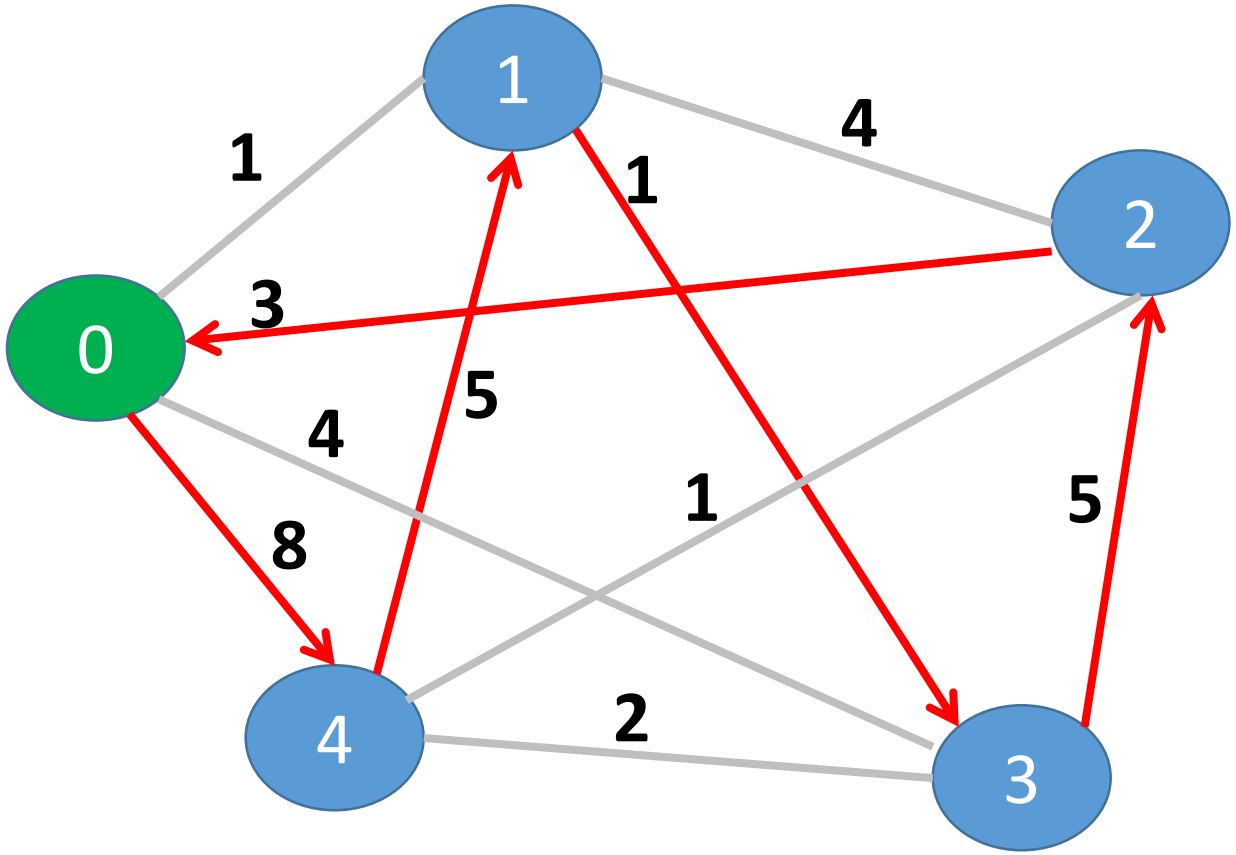
Cada elemento de la lista tabú puede ser un par de valores $(i, j), i \neq j$. Si el elmento (i, j) forma parte de la lista tabú, significa que ese enlace, entre las ciudades i y j ya fue eliminado de una solución para agregar otro enlace y modificar la solución existente. Por lo tanto ese enlace no podrá ser eliminado nuevamente durante un número de iteraciones, el cual es especificado como un parámetro del problema.

Representación de cada solución

Cada solución es una secuencia ordenada de ciudades, cuya posición en la lista indica el orden en el cual es visitada. Se considera que el último enlace une la última ciudad de la secuencia con la primera.

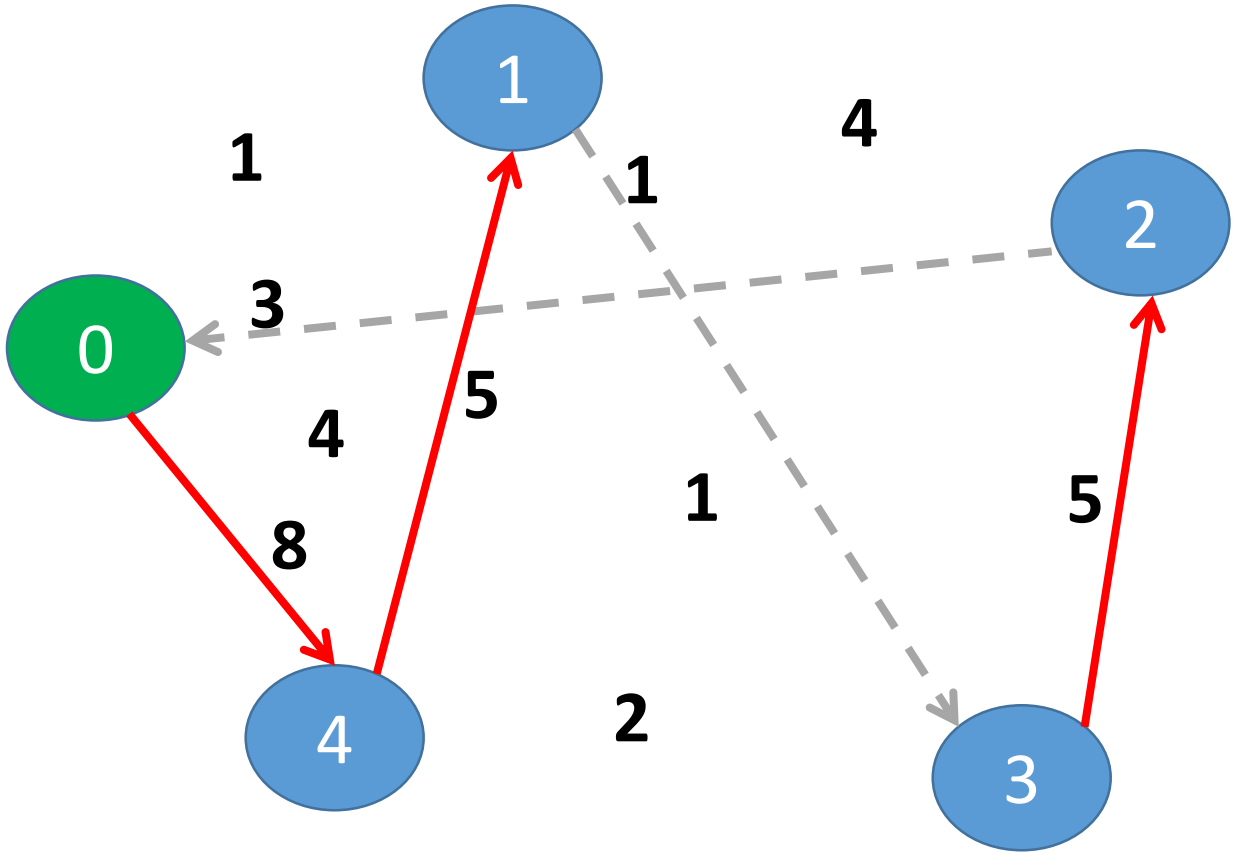
041320

Coste: 22



041320

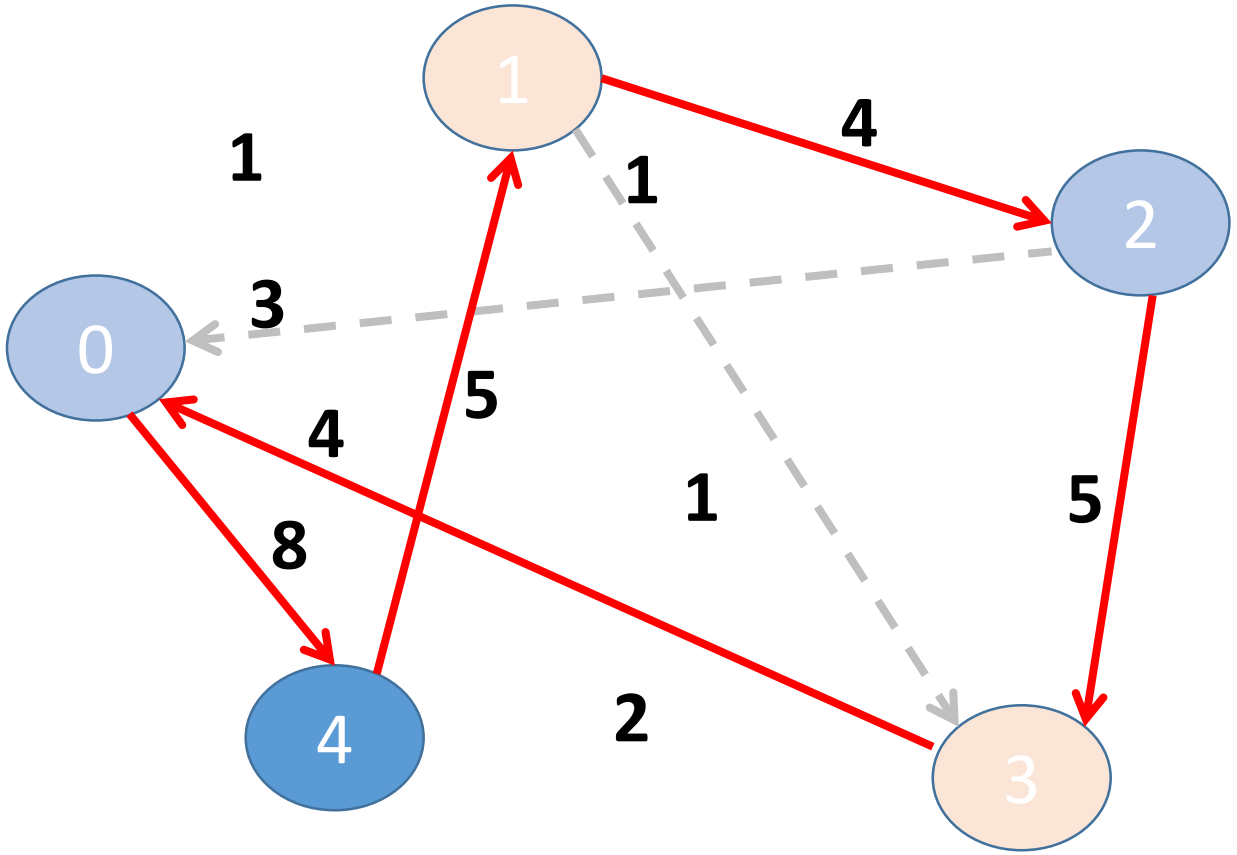
Coste: 22



Lista Tabú

(1,3)
(2,0)

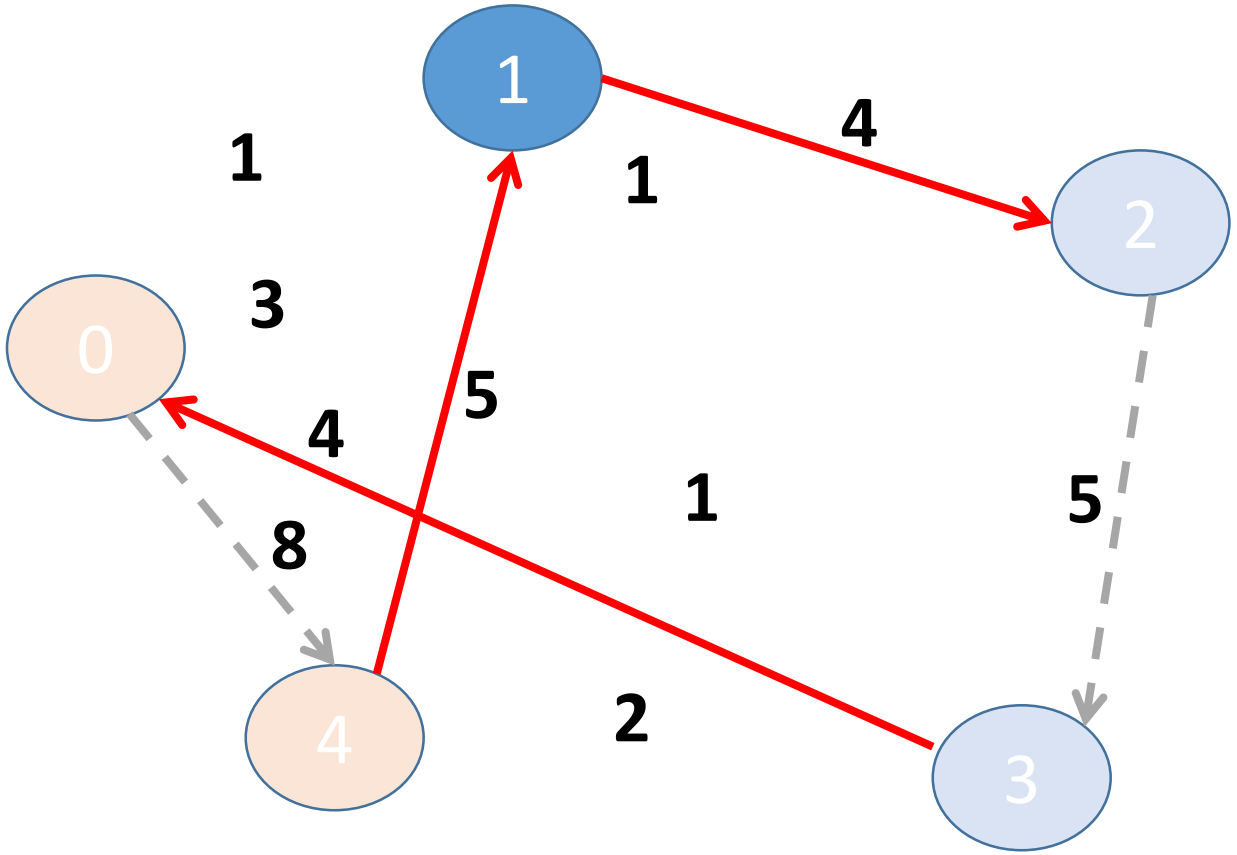
041230
Coste: 26



Lista Tabú

(1,3)
(2,0)

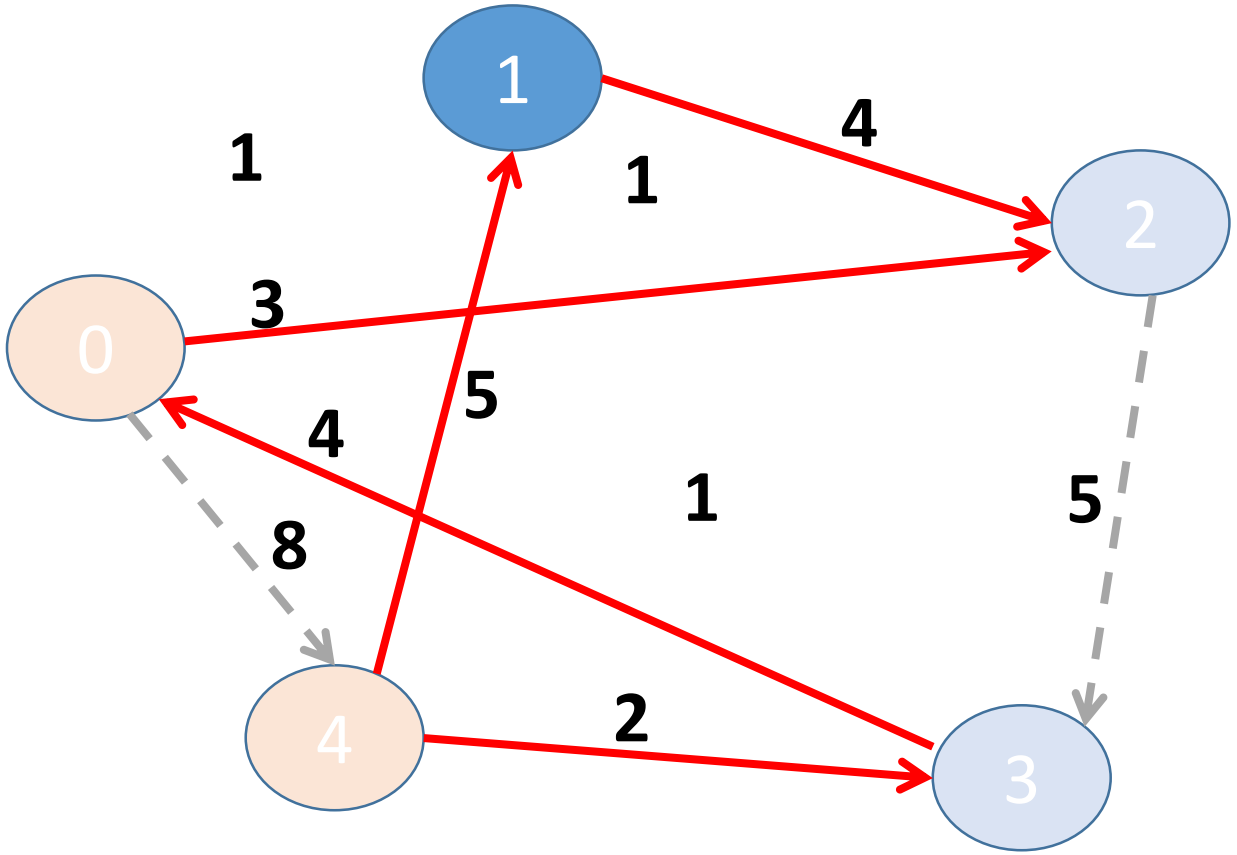
041230
Coste: 26



Lista Tabú

(1,3)
(2,0)

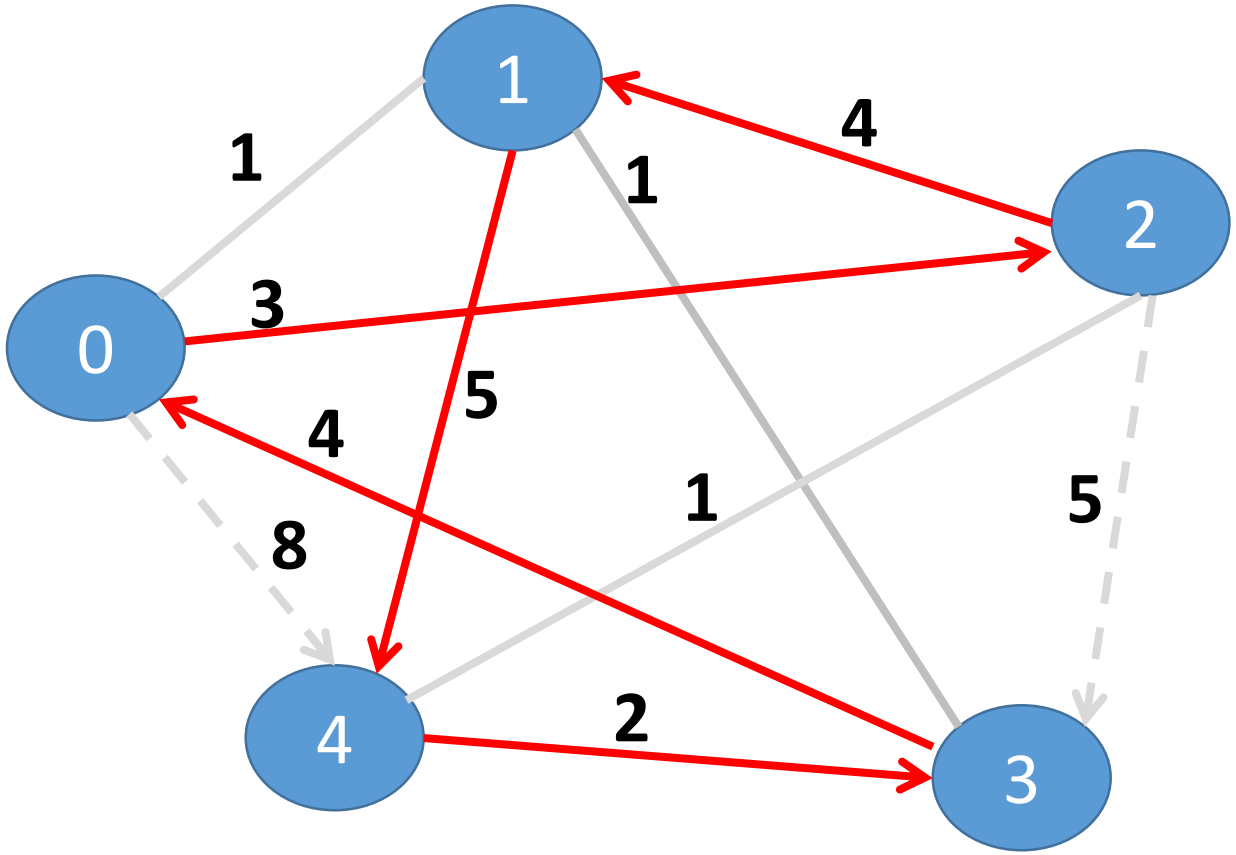
041230
Coste: 26



Lista Tabú

(1,3)
(2,0)
(2,3)
(0,4)

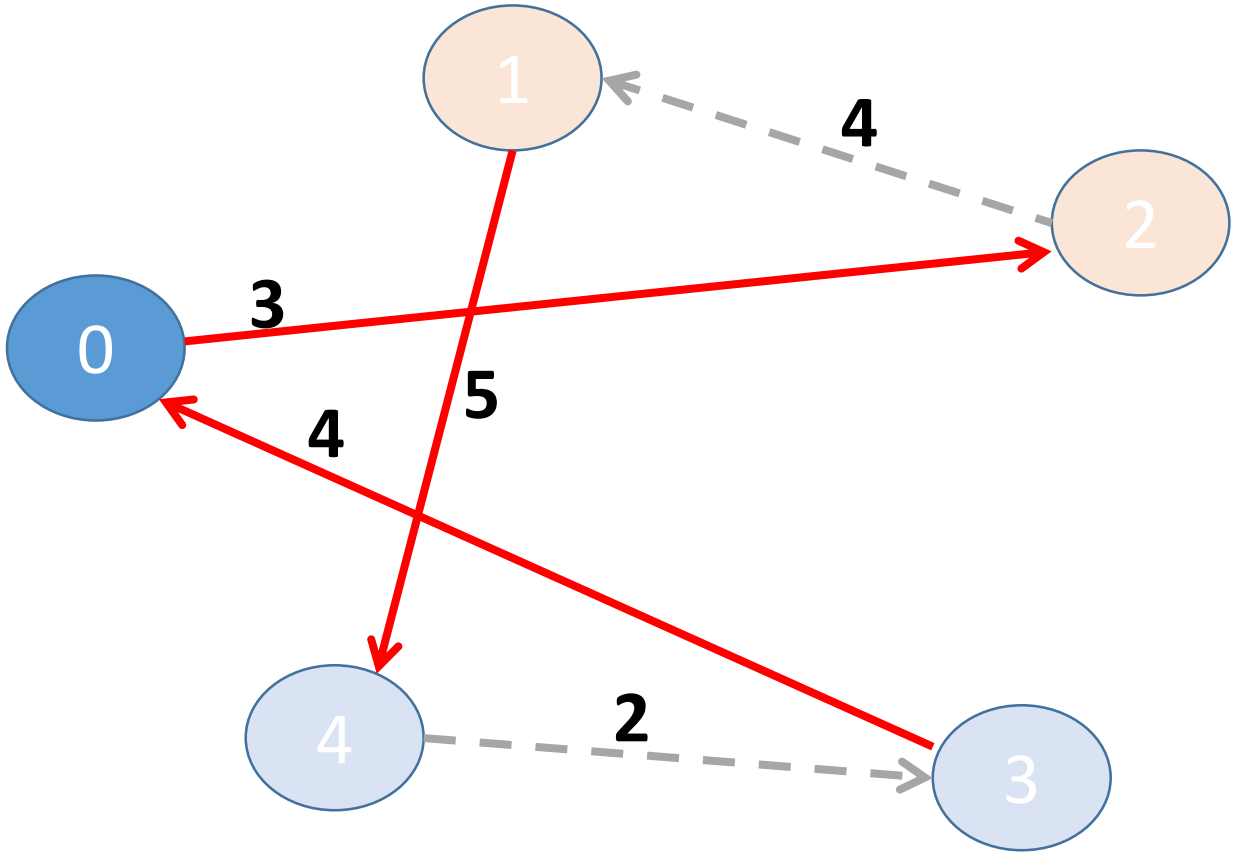
021430
Coste: 18



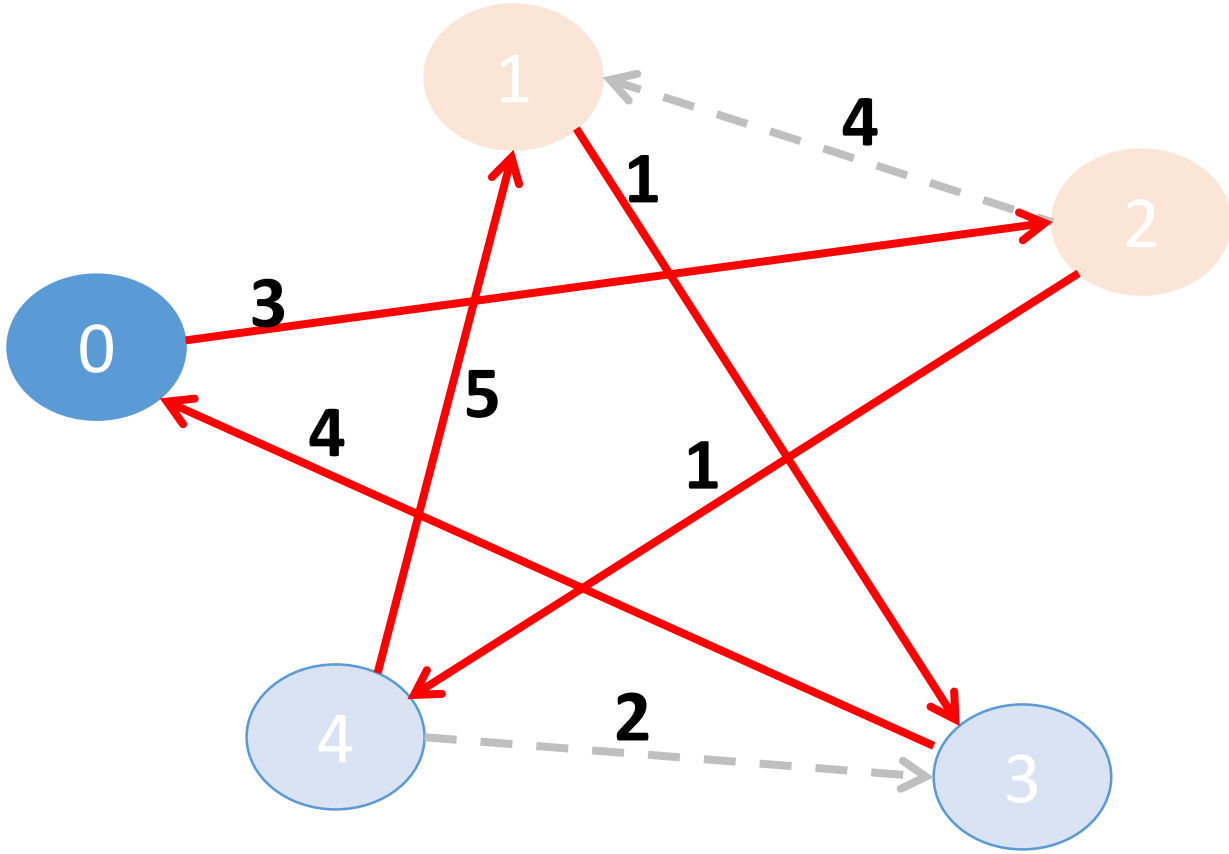
Lista Tabú

(1,3)
(2,0)
(2,3)
(0,4)

021430
Coste: 18



024130
Coste: 14

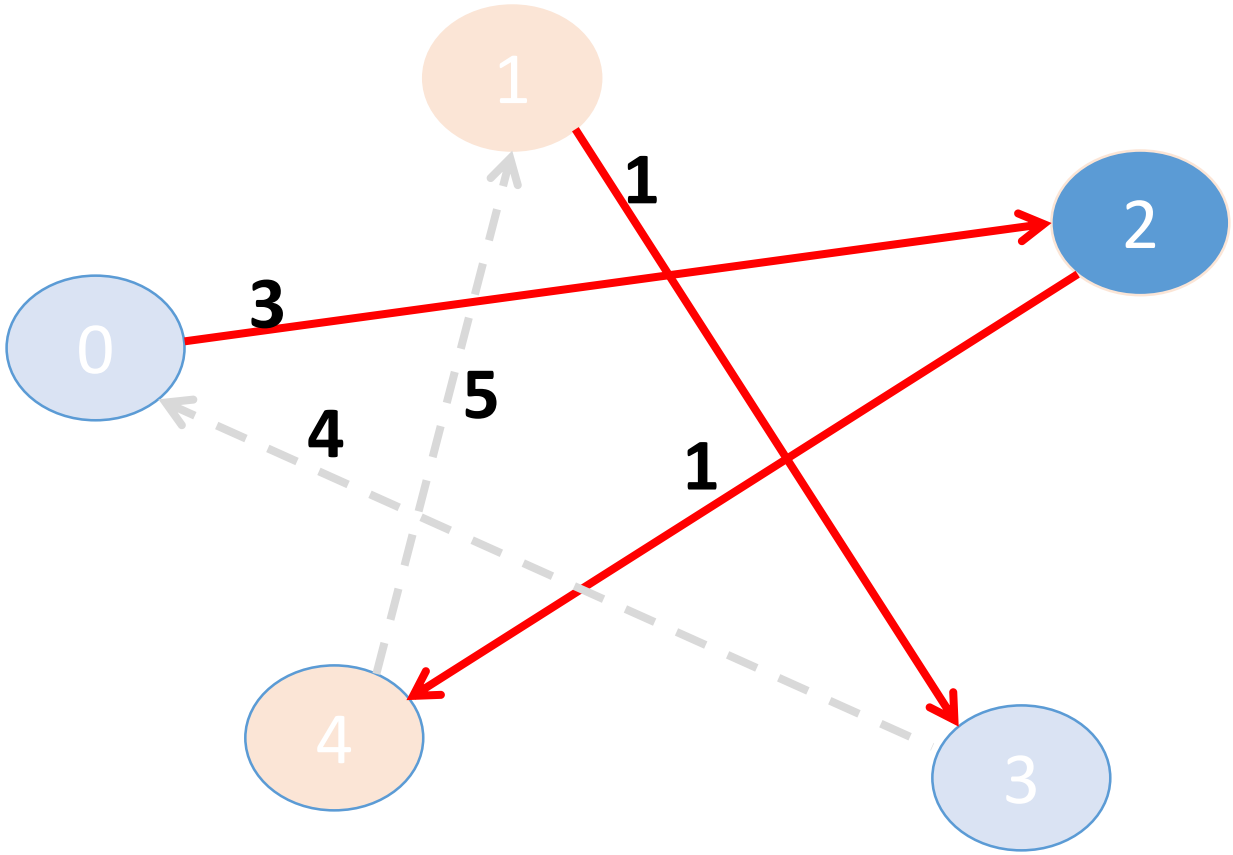


Lista Tabú

(1,3)
(2,0)
(2,3)
(0,4)
(2,1)
(4,3)

024130

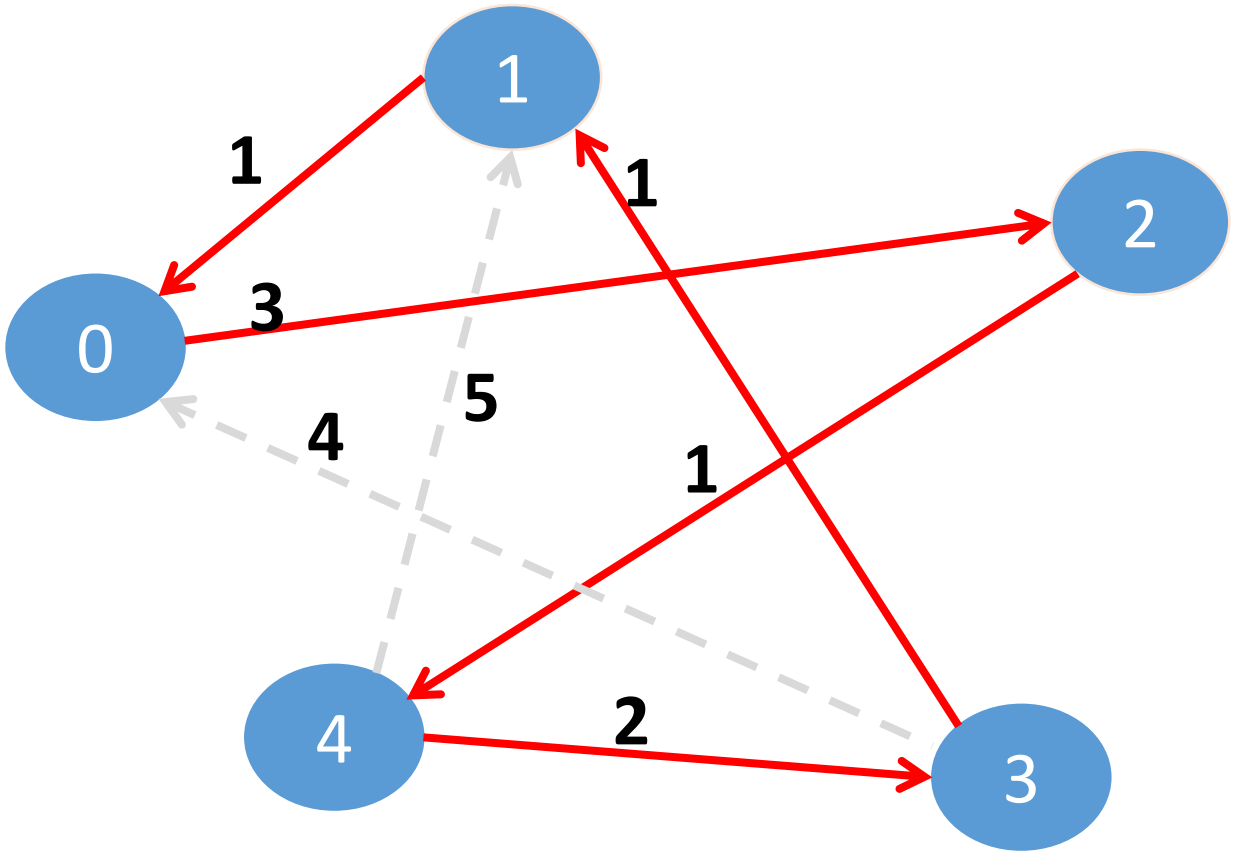
Coste: 14



Lista Tabú

(1,3)
(2,0)
(2,3)
(0,4)
(2,1)
(4,3)
(4,1)
(3,0)

024310
Coste: 9



Lista Tabú

(1,3)
(2,0)
(2,3)
(0,4)
(2,1)
(4,3)
(4,1)
(3,0)