IN7605 Heurísticas para Optimización Entera y Aplicaciones

Clase 2: Heurísticas: construcción y busquedas locales

Gonzalo Muñoz, Fernando Ordóñez

18 Agosto, 2025

Temário

Heurísticas tienen dos grandes piezas: encontrar soluciones factibles (de construcción) y mejorar las soluciones encontradas (de busqueda local).

- Heurísticas de Construcción
- Busqueda Local
- Ejemplos heurísticas para Timetabling

Si esto no es suficiente para resolver el problema de forma eficiente, se pueden utilizar métodos metaheurísticos o de descomposición.

Construcción: Enumeración

En algunos casos se pueden enumerar todas las soluciones y evaluar su factibilidad y calidad. Por ejemplo existen 2^5 soluciones $(x_1, x_2, x_3, x_4, x_5) \in \{0,1\}^5$ en:

$$
\begin{array}{lllllll}
\max & 9x_1 & +5x_2 & +7x_3 & +3x_4 & +x_5 & \\
\text{s.a} & 4x_1 & +3x_2 & +5x_3 & +2x_4 & +x_5 & \leq 10 \\
 & 4x_1 & +2x_2 & +3x_3 & +2x_4 & +x_5 & \leq 7 \\
 & x_1, & x_2, & x_3, & x_4, & x_5 & \in \{0,1\}
\end{array}
$$

Usar árbol binário: nodo raiz: separa soluciones con $x_1 = 0$ o $x_1 = 1$, 2do nivel: separa soluciones con $x_2 = 0$ de $x_2 = 1$, etc. Mejoras: cortar ramas infactibles, programa de forma recursiva

Construcción: Branch and Bound

Un MIP por un periodo de tiempo entrega una solución y una cota.

Input: A problem with n variables $x_1, \ldots, x_n$, policy α for managing sub-problems, relaxation method β, branching method γ

Result: An optimal solution x^* of value f^*

```
f* ← −∞                                   // Value of best solution found
F ← ∅                                     // Set of fixed variables
L ← {x_1, ..., x_n}                       // Set of free variables
Q ← {(F, L)}                              // Set of sub-problems to solve
while Q ≠ ∅ do
    Remove a problem P = (F, L) from Q according to policy α
    if P can potentially have feasible solutions with values already fixed in F then
        Compute a relaxation x of P with method β, modifying only variables x_k ∈ L
        if x is feasible for the initial problem and f* < f(x) then Store the improved
          solution
            x* ← x
            f* ← f(x)
        else if f(x) > f* then Expand the branch
            Choose x_k ∈ L according to policy γ
            forall possible value v of x_k do
                Q ← Q ∪ {(F ∪ {x_k = v}, L \ {x_k})}
        else No solution better than x* can be obtain
            Prune the branch
```

Construcción: Branch and Bound

Donde se deben definir

- **Métodos de relajación** β
  1. Relajación lineal
  2. Agregación de restricciones

$$\begin{array}{lrrrrrl} \max & 9x_1 & +5x_2 & +7x_3 & +3x_4 & +x_5 & \\ \text{s.a} & 8x_1 & +5x_2 & +8x_3 & +4x_4 & +2x_5 & \leq 17 \\ & x_1, & x_2, & x_3, & x_4, & x_5 & \in \{0,1\} \end{array}$$

- **Métodos de brancheo** γ
  - Si variable $x_k \in \{i, i+1, \ldots, i+k\}$ agregar $k+1$ problemas a Q con cada valor posible de x_k.
  - Si la variable x_k toma el valor continuo y. Agregar dos problemas a Q uno con $x_k \leq \lfloor y \rfloor$ y otro con $x_k \geq \lceil y \rceil$ (sin fijar variable x_k).
- **Métodos de gestión de subproblemas** α
  La lista Q se puede administrar: 1) una fila FIFO (breadth-first-search), 2) un STACK (depth-first-search), 3) una fila con prioridades (según estimación de valor óptimo)

Ejemplo de BnB para TSP

Esto necesita:

- Solución fija hasta depth y conjunto nodos libres L
- Se puede construir una cota inferior si el nodo en depth y el nodo 1 se pueden conectar a su nodo más cerca en L.
- Cada nodo en L se conecta con su nodo mas cercano en L.

El método trata de determinar el menor costo para parte del tour que aun está libre (L) considerando estructuras que no son rutas

Ejemplo de BnB para TSP

```python
from tsp_lower_bound import tsp_lower_bound                    # Listing 4.1

######### Basic Branch & Bound for the TSP
def tsp_branch_and_bound(d,                                  # Distance matrix
                         depth,  # current_tour[0] to current_tour[depth] fixed
                         current_tour,              # Solution partially fixed
                         upper_bound):                   # Optimum tour length

    n = len(current_tour)
    best_tour = current_tour[:]
    for i in range(depth, n):
        tour = current_tour[:]
        tour[depth], tour[i] = tour[i], tour[depth]
        lb, tour, valid = tsp_lower_bound(d, depth, tour)
        if (upper_bound > lb):
            if (valid):
                upper_bound = lb
                best_tour = tour[:]
                print("Improved: ", upper_bound, best_tour)
            else:
               best_tour, upper_bound = tsp_branch_and_bound(d, depth+1, tour, \
                                                      upper_bound)
    return best_tour, upper_bound
```

Construcción: Aleatória

- Fácil, difícilmente entregará buena solución
- Útil en métodos con busquedas locales posteriormente
- Es difícil generar soluciones on distribución uniforme

Input: A set of n elements $e_1, \ldots, e_n$
Result: A permutation p of the elements

```
i ← 0                                                  // Number of element already chosen
while i ≠ n do
    Draw a random number u uniformly between 1 and n
    if e_u is not already chosen then
        i ← i + 1
        p_i ← e_u
```

Construcción: Golosa

En una heurística golosa se busca construir una solución s gradualmente, agregando el elemento e a la solución de forma de obtener el mejor costo parcial $c(s, e)$ en cada paso.

Input: A trivial partial solution s (generally $\varnothing$); set E of elements constituting a solution; incremental cost function $c(s, e)$
Result: Complete solution s

```
R ← E                                         // Elements that can be added to s
while R ≠ ∅ do
    ∀e ∈ R, compute c(s, e)
    Choose e′ optimizing c(s, e′)
    s ← s ∪ e′                                 // Include e′ in the partial solution s
    Remove from R the elements that cannot be added any more to s
```

Construcción Golosa: TSP

- Golosa (en arcos): $c(s,(jk)) = d_{jk}$
- Vecino más cercano: $c(s,e) = d_{ie}$
- Máximo arrepentimiento: $c(s,e) = \min_{j,k\in R} d_{je} + d_{ek} - \min_{j\in R} d_{ie} - d_{ej}$
- Menor/Mayor inserción: $c(s,e) = \min_{j\in s} d_{je} + d_{ej+1} - d_{jj+1}$

Construcción Golosa: Coloreo de Grafos

Métrica: saturación del nódo (i.e. no. colores en nodos adjacentes) Esta se define: 1) no. colores distintos en nodos adjacentes, 2) grado del nodo
Se busca colorear nodos con mayor saturación primero.

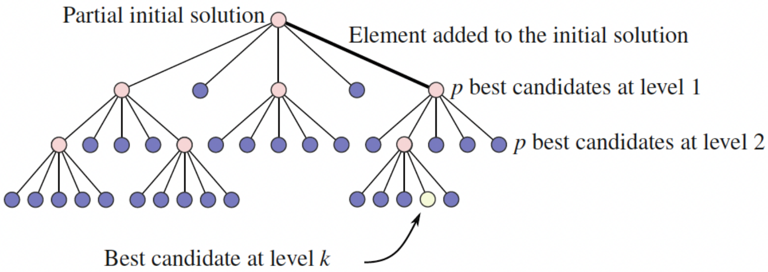
Input: Undirected graph $G = (V, E)$;
Result: Vertex coloring

Color with 1 the vertex v with the highest degree
$R \leftarrow V \setminus v$
$colors \leftarrow 1$
while $R \neq \varnothing$ **do**
    $\forall v \in R$, compute $DS(v)$
    Choose v' maximizing $DS(v')$, with the highest possible degree
    Find the smallest k $(1 \leqslant k \leqslant colors + 1)$ such that color k is feasible for v'
    Assign color k to v'
    if $k > colors$ **then**
        $colors = k$
    $R \leftarrow R \setminus v'$

Mejoras a construcciones golosa

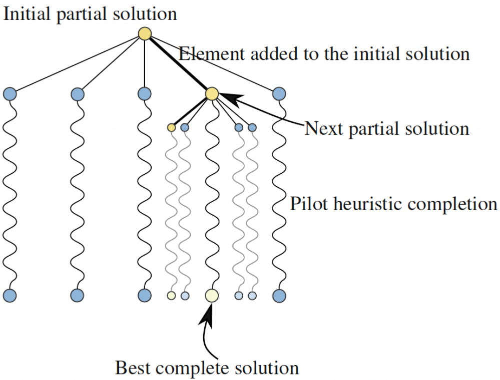
Presentamos dos metodos que integran enumeración parcial como una forma de mejorar una heurística golosa.

- **Busqueda por viga**
  Se redefine el $c(s, e)$ usando enumeración parcial (p soluciones por etapa) durante k etapas.



Mejoras a construcciones golosa

- **Metodo piloto**
  - Usa una heurística piloto para evaluar costo de agregar e a s.
  - El e que genere la mejor solución con la heurística piloto partiendo de s mas e será el siguiente elemento generado.
  - ¿Esto puede entregar una solución distinta a lo que da la heurística piloto?



Busqueda Local

Idea general:

1. Partir con una solución factible s.
2. Examinar soluciones s' en una vecindad $N(s)$ de s.
3. Moverse a una buena solución $s' \in N(s)$.
4. $s = s'$ y registrar si s es lo mejor obtenido, ir a 2.

Criterios de parada:

- Tiempo del algoritmo.
- Número de iteraciones.
- No se mejora la mejor solución en un cierto número de iteraciones.
- Valor de la función objetivo.

Busqueda Local

La busqueda local se dice

- Primera mejora: si se acepta el primer $s' \in N(s)$ tal que $f(s') < f(s)$. Como un depth-first search.
- Máxima mejora: si se acepta $s' = \mathrm{argmin}_{u \in N(s)} f(u)$ tal que $f(s') < f(s)$.

En ambos casos se termina con un óptimo local, relativo a $N(s)$.

Es crucial la definición de $N(s)$ para el éxito de busqueda local. Notación:

- $N(s)$ conjunto de soluciones que son vecinas de la solución s.
- $M(s)$ movimientos (modificaciones) de s que transforman s en uno de sus vecinos.
- $s \oplus m$ aplicar el movimiento m a la solución s. Con esto tenemos

$$N(s) = \{s \oplus m \mid m \in M(s)\} .$$

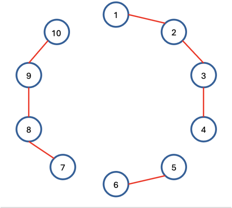
Busqueda Local: caso TSP

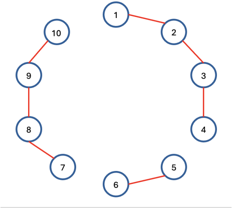
Veremos algunos ejemplos de vecindades para TSP:

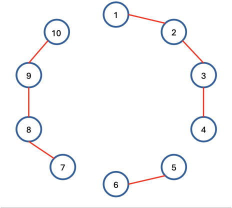
- **2-Opt:** Para una solución (tour) s, un movimiento $[i, j]$ de 2-Opt requiere
  - El tour s se mueve (i, s_i) y (j, s_j).
  - $i \neq j$, $i \neq s_j$, $j \neq s_i$
  - El tour $s' \in N(s)$ borra los arcos (i, s_i), (j, s_j) cambiandolos por (i, j) y (s_i, s_j)
  - Nota que los nodos $s_i, s_i + 1, \ldots, j$ ahora se visitan en dirección opuesta
  - Tamaño de la vecindad: $O(n^2)$
  - Cambio en costo (caso simétrico) $= c_{ij} + c_{s_i,s_j} - c_{i,s_i} - c_{j,s_j}$

Busqueda Local: caso TSP

- **3-Opt:** Eliminar tres arcos $(i, s_i), (j, s_j), (k, s_k)$ y reconstruir un tour. ¿De cuantas formas se puede reconstruir un tour?







Busqueda Local: caso TSP

La siguiente tabla muestra el número de formas de recomponer una ruta con un 2-Opt, 3-Opt, etc.

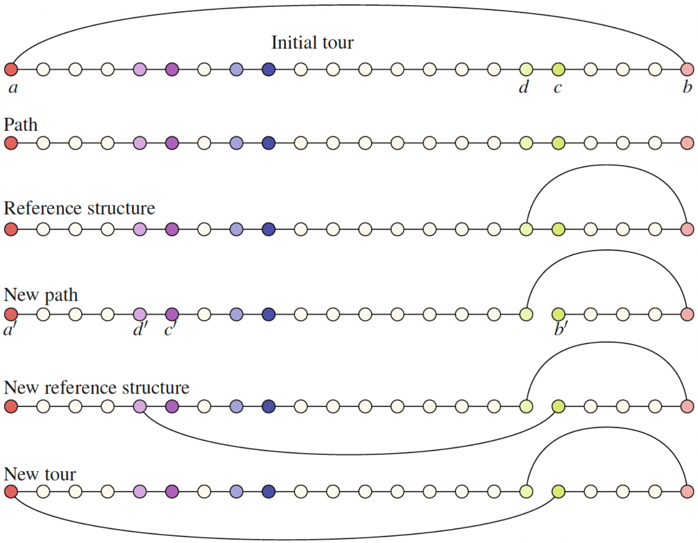
k	# rutas
2	1
3	4
4	20
5	148
6	1368
7	15104

Tamaño de la vecindad 3-Opt?

Busqueda Local: caso TSP

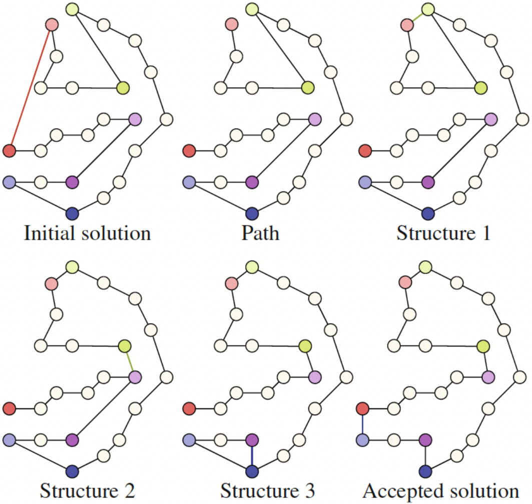
- **Lin-Kernigan:** Es un ejection chain (vecindad mayor construida con una secuencia de movimientos)
  - una ejección es borrar un arco y agregar otro conectado con uno de los nodos libres
  - una solución *s* se transforma a una estructura de referencia
  - la estructura de referencia se puede transformar en una solución o una estructura de referencia
  - se repite hasta que se encuentra una solución mejor

Busqueda Local: caso TSP



Busqueda Local: caso TSP

Un ejemplo de Lin-Keringan en rutas



Busqueda Local: Propriedades de Vecindades

Algunas propriedades deseables de las vecindades $N(s)$

- Conectividad: De cada solución s debe haber una secuencia de cambios que llegue a una solución global. La secuencia puede no ser monótona en la función objetivo.
- Diametro pequeño: Se debe poder llegar a la solución óptima en pocos pasos.
- Baja rugosidad: Debe tener pocos mínimos locales. Una forma de alisar una vecindad es el método del *elefante volador*, cambiando la función objetivo dado un parámetro $\tau > 0$ de la siguiente forma

$$|x| \sim \sqrt{x^2 + \tau^2} \qquad \max\{0, x\} \sim (x + \sqrt{x^2 + \tau^2})/2$$

Busqueda Local: Propiedades de Vecindades

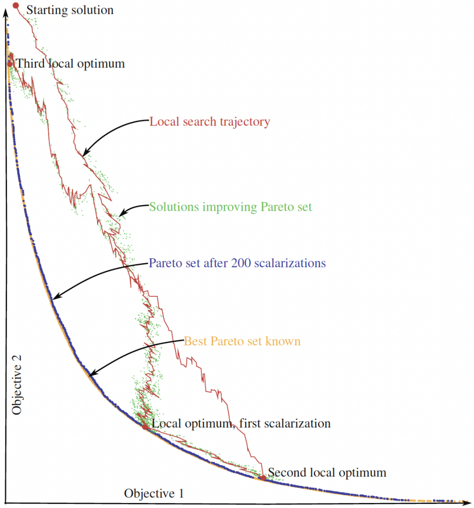
- Tamaño pequeño: Se debe poder revisar la vecindad rápidamente
- Evaluación rápida: Se debe poder evaluar soluciones en la vecindad rápidamente. Por ejemplo, en el TSP asimetrico ya no es posible evaluar una solución en la vecindad 2-opt en tiempo constante.

En general no se tienen todas estas propiedades y existe mucho trabajo en como manejar vecindades que no se portan bien: son grandes, de difícil evaluación, no son conectadas, etc.

Busqueda Local: Multi-objetivo

- **Escalamiento**

  - Utiliza pesos w_i para cada objetivo i para combinar los objetivos linealmente en un sólo objetivo
  - Para casa peso w, usa heurística de mejoramiento para encontrar solución cerca del conjunto Pareto
  - Genera $I_{\max}$ pesos w para explorar el conjunto Pareto



Busqueda Local: Multi-objetivo

- **Busqueda Local Pareto**
  Método explora $N(s)$ vecindad de s, recursivamente, guardando soluciones no dominadas.

Neighborhood_evaluation
Input: Solution s; neighborhood $N(\cdot)$ objective functions $\vec{f}(\cdot)$
Result: Pareto set P completed with neighbors of s
forall $s' \in N(s)$ **do**
    Update_Pareto($s', \vec{f}(s')$)

Update_Pareto
Input: Solution s, objective values $\vec{v}$
Result: Updated Pareto set P
if $(s, \vec{v})$ *either dominates a solution of* P *or* $P = \varnothing$ **then**
    From P, remove all the solutions dominated by $(s, \vec{v})$
    $P \leftarrow P \cup (s, \vec{v})$
    Neighborhood_evaluation(s)

Busqueda Local: Multi-objetivo

- **Estructura de Datos**
  En estos métodos se revisa si una solución es o no dominada por el conjunto Pareto. Dado que el conjunto de soluciones no dominadas puede ser grande, es necesario una estructura de datos para almacenar el conjunto pareto:
  - En el caso con dos objetivos. Si un objetivo es entero y está acotado entre k y $k+l$, basta un arreglo A de largo l, donde $A[i]$ tiene el mejor valor del otro objetivo entre k y $k+i$. Por ejemplo: si tenemos las soluciones $(2,27),(4,24),(6,23),(7,21),(11,17)$, y el primer objetivo está entre 2 y 13, lo podemos almacenar en el siguiente arreglo de largo $13-2+1=12$:
    $$[27,27,24,24,23,21,21,21,21,17,17,17]$$
    ¿Como determina si una solución $(5,20)$ es dominada o no?
  - En el caso de K objetivos se puede hacer esto en un arreglo de dimensión $K-1$.
  - Árbol KD: Un árbol binário en que
    - un nodo en profundidad d discrimina según el objetivo $d \mod K$.
    - una busqueda en el árbol es rápida en el tamaño del conjunto P, crece como $\log(|P|)$
    - inserciones y eliminaciones del conjunto P son mas complicadas

Ejemplos Heurísticas para Timetabling

- Timetabling: asignación de eventos en ciertos horarios de tiempo, cada evento compromete recursos que llevan a
  - Restricciones duras.
  - Restricciones blandas.

- Ejemplos:
  - Programación de cursos: recursos son profesores, salas de clases, alumnos
  - Programación de torneos: recursos = equipos, horarios televisivos, local/visita
  - Programación de tareas a maquinas: recursos = maquinas

- Problema de gran tamaño, dificil (o imposible de resolver a optimalidad).

Timetabling: Modelo y Supuestos

Estructura

- *timeslots*: $T_1, T_2, \ldots, T_{45}$.
- Salas: $R_1, R_2, \ldots, R_{N_R}$.
- *events* E_j (conjunto de estudiantes S_j): $E_1, E_2, \ldots, E_{N_E}$.
- *room features*: $F_1, F_2, \ldots, F_{N_F}$.
- Cada sala posee ciertos *features*, como cada *evento* requiere ciertos *features*.
- Se define un *place* P como:

$$P = (T_k, R_j).$$

Timetabling: Restricciones y F.O.

Restricciones duras

- Ningún alumno atiende a más de una clase.
- Ningún *event* es asignado a más estudiantes de los que caben en la sala asignada.
- Todos los requerimientos de un *event* son cumplidos por la sala.
- Nungún *place* tiene más de un evento asignado.

Restricciones blandas (penalizaciones)

- Un alumno tiene clases en el último *timeslot* de un día.
- Un alumno tiene más de dos clases seguidas.
- Un alumno tiene en un día una sola clase.

Heurísticas: Construcción

1. Lista L con todos los *events*.
2. Tomar $E \in L$ con el mínimo de *places* posibles.
3. Para cada place P calcular $q = (q_1, q_2, \ldots, q_5)$, y tomar el P que minimice $w^T q$. Donde:
   $q_1 =$ Núm. de *events* no asignados, asignables en P.
   $q_2 =$ Núm. de salas pedidas en el mismo *timeslot* que P.
   $q_3 =$ Núm. de restr. blandas 1 creadas al asignar E a P.
   $q_4 =$ Núm. de restr. blandas 2 creadas al asignar E a P.
   $q_5 =$ Núm. de restr. blandas 3 creadas al asignar E a P.
4. Asignar E a *place* P. Si $L = \phi$, terminar, si no volver a 2.
5. Si falla el proceso de 1-4 (*events* sin *place*), se parte de nuevo con otra semilla aleatoria.

Heurísticas: Busquedas locales

Idea general de Busqueda Local:

1. Partir con una solución factible s.
2. Examinar soluciones s' en una vecindad $N(s)$ de s.
3. Moverse a una buena solución $s' \in N(s)$.
4. $s = s'$ y registrar si s es lo mejor obtenido, ir a 2.

Heurísticas: Busquedas locales

En particular en este ejemplo podemos:

- Tomar $N(s) =$ cambio de un solo *event* con respecto a s.
  Cambiar la asignación de un conjunto de estudiantes S_j de un place P a otro.

- Moverse a una buena solución $s' \in N(s)$.
  a) Para cada E, escoger P con mejor valor.
  b) Enlistarlos (una movida por *event*).
  c) Ordenar primero por valores, y luego por frecuencia.
  d) Si la mejor opción es mejor al menor valor obtenido, escoger esa. Si no, en los no-TABU, escoger con una VA $geom(\frac{1}{2})$.

Busquedas locales

Otras vecindades $N(s)$ se encuentran trabajando con la formulación entera del problema

- Se fijan ciertos aspectos de la solución a lo que da s
  - Todas las asignaciones de S_j para $j \neq k$ se mantienen
  - Se mantienen asignaciones en las salas R_i con $i \neq k$
  - Se mantienen asignaciones en periodos T_h con $h \neq k$
- El resto de las variables se permite que sea libre
- La busqueda por el mejor vecino consiste en resolver el problema reducido (tipicamente mas facil).
- Se conoce como "hacer un dive" de la solución s

Timetabling, solo classes a salas: Modelo

- Se desean asignar clases en respectivos modulos de tiempo, en ciertas salas que cumplan los requerimientos adecuados de material y tamaño.

Consideraciones:

- $0 << \text{alumnos} \,/\, \text{tamaño de la sala} \leq 1$
- Ubicación de la sala de clases.
- Mejor tener clases consecutivas.
- Nueve bloques de 1 hora y seis bloques de 1.5 horas.
- Bloques de hora prime (pedir max 60% por dpto).

Timetabling, solo classes a salas: Heurística

1. Seleccionar slot t con menor ratio oferta/demanda.
2. Tomo clases $j \in J_t$ de mayor a menor número de alumnos, y las asigno a sala M_j de menor costo.
3. Ordeno clases $j \in J_t$ en orden decreciente en costo desde el costo actual y para cada una, si j:
   - No está asignada, veo si la puedo introducir en una sala asignada y mover esa clase a otra sala. Tomando el movimiento que reduce el costo en mayor medida.
   - Si está asignada, veo entre todos los swaps con otra clase (y sus salas), y tomo aquel que reduce en más los costos.
4. Si en el paso 3 se reducieron los costos, volver a el, si no borrar de la lista de clases aquellas recién asignadas.
   Si todos los slots de tiempo han sido revisados, terminar.

Timetabling torneo todos contra todos: Modelo

- single round robin = todos contra todos 1 vez
  (campeonato de apertura)
- n equipos
- si n par todos contra todos juegan en $n-1$ turnos
- n impar $\Rightarrow$ n turnos en cada turno 1 descanza
  (clasificatorias Brasil 2014)
- en cada turno hay $\lfloor n/2 \rfloor$ juegos
- se debe decidir quien juega con quien en cada turno
- quien es local y visita

Timetabling todos v. todos: Formulación

$$\begin{array}{ll} \sum_{i=1}^{n}(x_{ijt} + x_{jit}) = 1 & j = 1 \ldots n, t = 1 \ldots n-1 \\ \sum_{t=1}^{n-1}(x_{ijt} + x_{jit}) = 1 & i, j = 1 \ldots n, i \neq j \end{array}$$

funciones objetivos:

- reducir quiebres: ningun equipo puede tener mas de dos L o V seguidos
- programación horaria: ningun equipo puede jugar mas de 2 veces en un horario
- reducir tiempo de viaje

Todos v. todos: Patrones

- Soluciones se pueden ver como sequencias de $n-1$ local y visita
  - $n=6$: LLVVV; LVLVL;
  - $n=5$: LLBVV; LBVLV;
- Para tener sentido en cada turno debe haber el mismo número de L y de V
- Una solución corresponde a un coloreo óptimo de los arcos del grafo completo
  - nodos son los equipos, arcos todos contra todos
  - n par (impar) nodos existe un coloreo de arcos con $n-1$ (n) colores
  - cada turno corresponde a un color diferente.

Todos v. todos: Heur. Constructiva

1. Juntar n patrones de locales y visitas (PLVs)
2. Asignar un juego a cada evento en el conjunto de PLVs
3. Asignar un equipo a cada PLV

Ejemplo

$$\begin{array}{ll} \text{eq } 1 & VLV \\ \text{eq } 2 & LVL \\ \text{eq } 3 & LLV \\ \text{eq } 4 & VVL \end{array} \Rightarrow \begin{array}{llll} \text{eq } 1 & 3 & -4 & 2 \\ \text{eq } 2 & -4 & 3 & -1 \\ \text{eq } 3 & -1 & -2 & 4 \\ \text{eq } 4 & 2 & 1 & -3 \end{array} \Rightarrow \begin{array}{llll} \text{eq d} & a & -c & b \\ \text{eq b} & -c & a & -d \\ \text{eq a} & -d & -b & c \\ \text{eq c} & b & d & -a \end{array}$$

Todos v. todos: Heurística Constructiva

Encontrar soluciones en cada paso puede requerir problemas de optimización. Por ejemplo

- Dado S conjunto de PLVs
- T conjunto de turnos
- $F \subset S \times S \times T$ combinaciones factibles si equipo de patron k puede jugar con equipo de patron l en periodo t
- encontrar juego a cada evento en este conjunto S es

$$
\begin{array}{lll}
\min & \sum_{(k,l,t)\in F} x_{klt} & \\
\text{s.a.} & \sum_{t:(k,l,t)\in F} x_{klt} + \sum_{t:(l,k,t)\in F} x_{lkt} = 1 & k,l \in S, k \neq l \\
 & \sum_{l:(k,l,t)\in F} x_{klt} + \sum_{l:(l,k,t)\in F} x_{lkt} \leq 1 & k \in S, t \in T \\
 & x_{klt} \in \{0,1\} & (k,l,t) \in F
\end{array}
$$

Todos v. todos: Busqueda Local

Vecindades para el problema que minimiza número de equipos que juegan más de 2 veces en un horario.

- dada una solución S (PLVs con asignación de equipos y de juegos a horarios)
- fijar la asignacion de equipos a PLVs
- permitir cambios de juegos en horarios sólo en 1 semana