



# INSTITUTO POLITÉCNICO NACIONAL

## ESCUELA SUPERIOR DE CÓMPUTO - IPN

### TÓPICOS SELECTOS DE ALGORITMOS BIOINSPIRADOS

# ALGORITMO GENÉTICO HÍBRIDO PROBLEMA DEL AGENTE VIAJERO

*Presenta*

**Dr. DANIEL MOLINA PÉREZ**

**danielmolinaperez90@gmail.com**

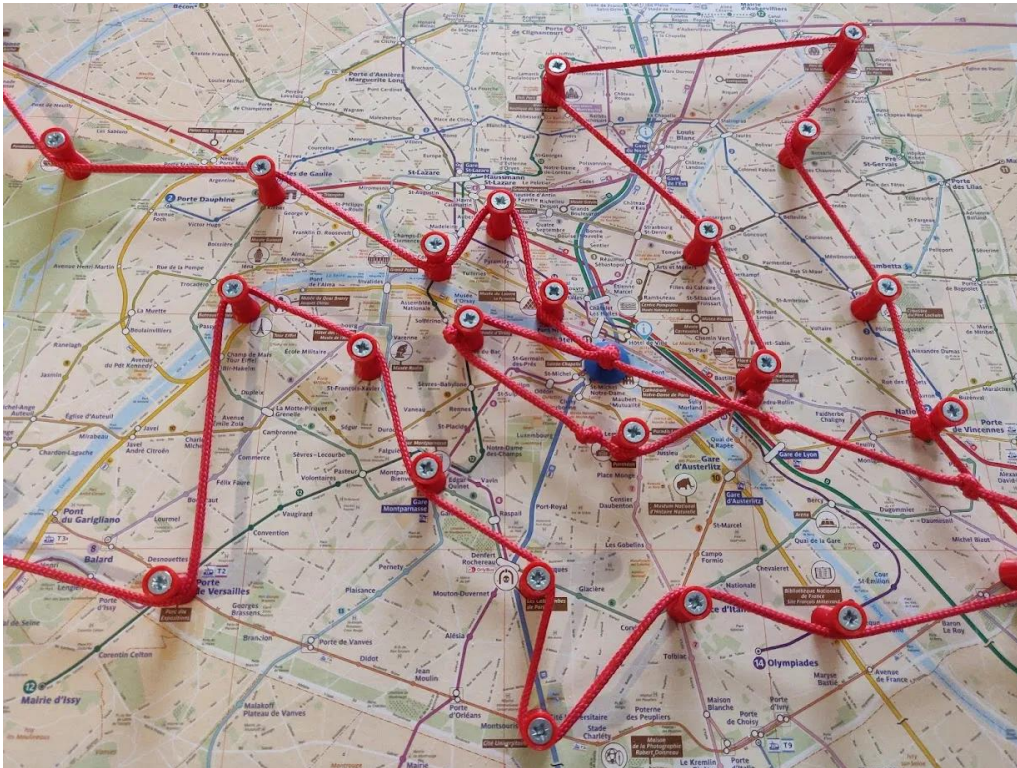
INSTITUTO POLITÉCNICO NACIONAL



ESCOM

CIUDAD DE MÉXICO

# PROBLEMA DEL AGENTE VIAJERO (TRAVELLING SALESMAN PROBLEM)



- El Problema del Agente Viajero (TSP) es un problema de optimización combinatoria que implica encontrar el camino más corto que visita cada ciudad exactamente una vez y regresa al punto de partida. Formalmente, dado un conjunto de ciudades y las distancias entre cada par de ciudades, el objetivo es encontrar la ruta de menor distancia que visite todas las ciudades y regrese al punto de partida.
- El TSP tiene una amplia gama de aplicaciones en la vida real, como la planificación de rutas de entrega, la optimización de recorridos de vehículos, la programación de circuitos electrónicos, la logística y la planificación de itinerarios turísticos. Debido a su relevancia práctica y desafíos computacionales, el TSP ha sido objeto de extensa investigación en el campo de la optimización combinatoria y la inteligencia artificial.

# PROBLEMA DEL AGENTE VIAJERO: REPRESENTACIÓN EN VARIABLES BINARIAS

Encontrar la ruta de menor distancia que visite todas las ciudades y regrese al punto de partida

Ir de la ciudad 1 a la 2 ¿si o no?

Ir de la ciudad 1 a la 3 ¿si o no?

Ir de la ciudad 1 a la 4 ¿si o no?

Ir de la ciudad 2 a la 1 ¿si o no?

Ir de la ciudad 2 a la 2 ¿si o no?

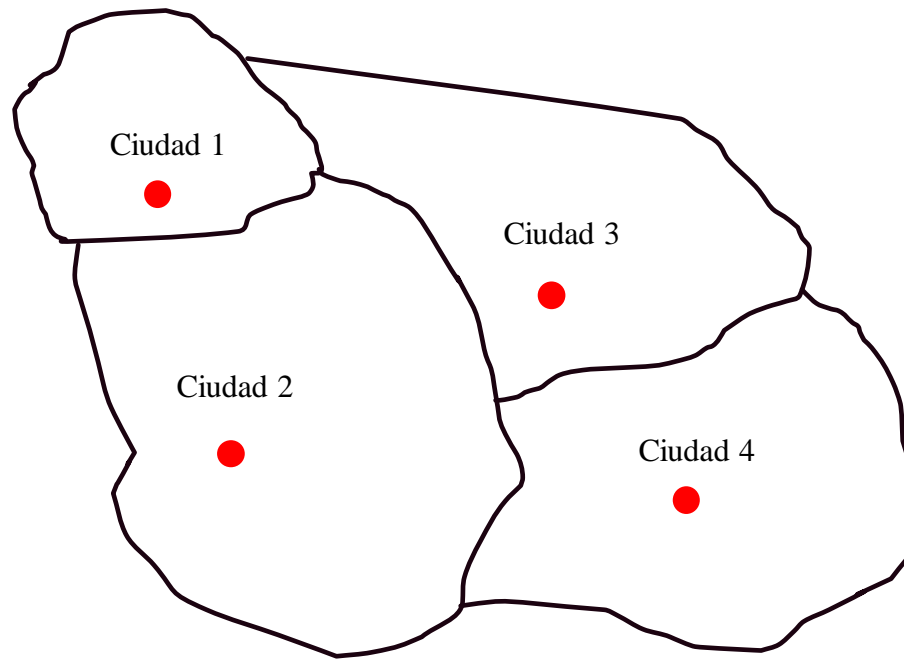
Ir de la ciudad 2 a la 3 ¿si o no?

Ir de la ciudad 2 a la 4 ¿si o no?

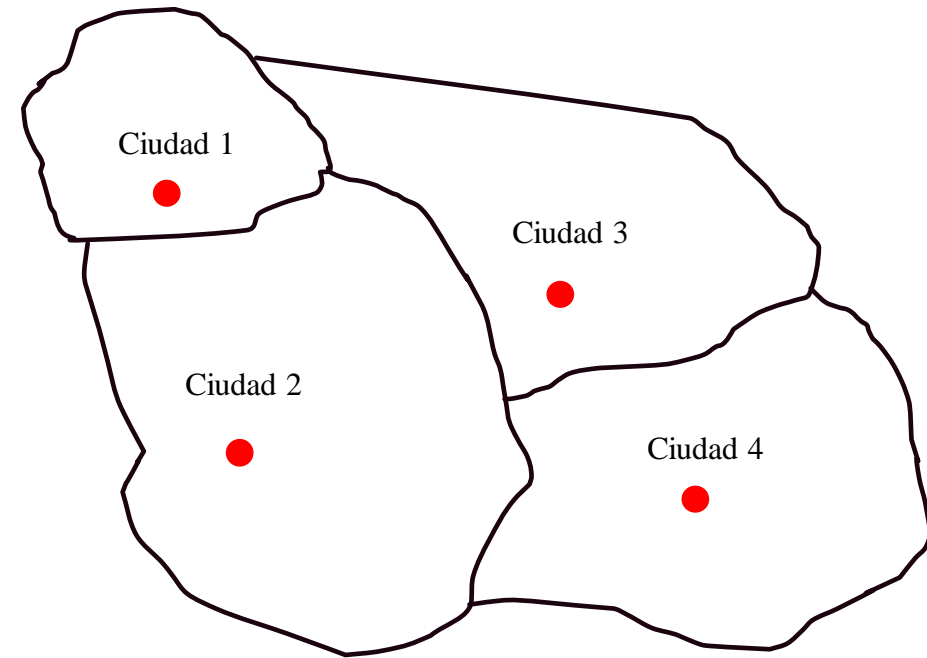
.

.

.



# PROBLEMA DEL AGENTE VIAJERO: REPRESENTACIÓN EN VARIABLES BINARIAS



La variable  $x_{ij}$  representa el trayecto entre la ciudad  $i$  y la ciudad  $j$ :

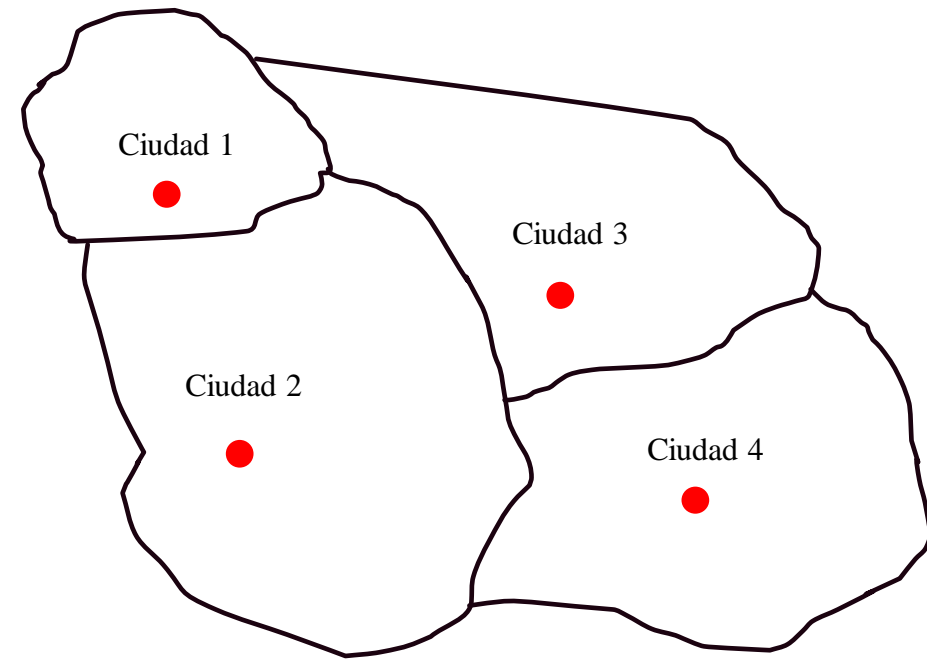
- $x_{ij} = 1$  si existe conexión (o arco) entre la ciudad  $i$  y la ciudad  $j$
- $x_{ij} = 0$  si **no** existe conexión (o arco) entre la ciudad  $i$  y la ciudad  $j$

Siendo el vector de variables para el problema de cuatro ciudades:

$$\mathbf{x} = [x_{1,2} \ x_{1,3} \ x_{1,4} \ x_{2,1} \ x_{2,3} \ x_{2,4} \ x_{3,1} \ x_{3,2} \ x_{3,4} \ x_{4,1} \ x_{4,2} \ x_{4,3}]$$

Este vector contiene todas las conexiones (o arcos) posibles, y tiene la condición que siempre  $i \neq j$ . Un problema de  $n$  ciudades contiene  $n(n - 1)$  variables

# PROBLEMA DEL AGENTE VIAJERO: REPRESENTACIÓN EN VARIABLES BINARIAS



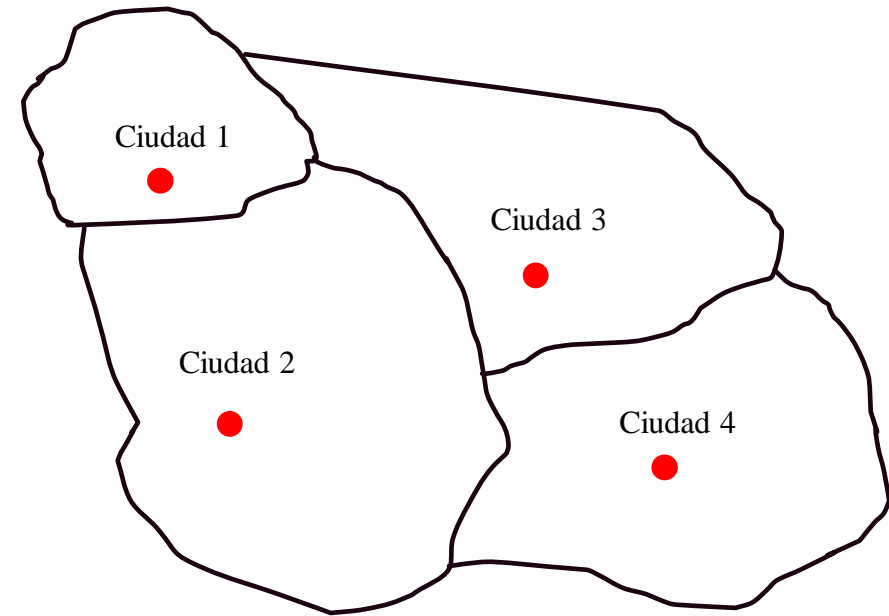
La variable  $x_{ij}$  representa el trayecto entre la ciudad  $i$  y la ciudad  $j$ :

- $x_{ij} = 1$  si existe conexión (o arco) entre la ciudad  $i$  y la ciudad  $j$
- $x_{ij} = 0$  si **no** existe conexión (o arco) entre la ciudad  $i$  y la ciudad  $j$

Considerando lo anterior dibuje las siguientes rutas:

$$\mathbf{x} = \begin{bmatrix} x_{1,2} & x_{1,3} & x_{1,4} & x_{2,1} & x_{2,3} & x_{2,4} & x_{3,1} & x_{3,2} & x_{3,4} & x_{4,1} & x_{4,2} & x_{4,3} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \end{bmatrix}$$

# PROBLEMA DEL AGENTE VIAJERO: REPRESENTACIÓN EN VARIABLES BINARIAS



Matriz de costos:

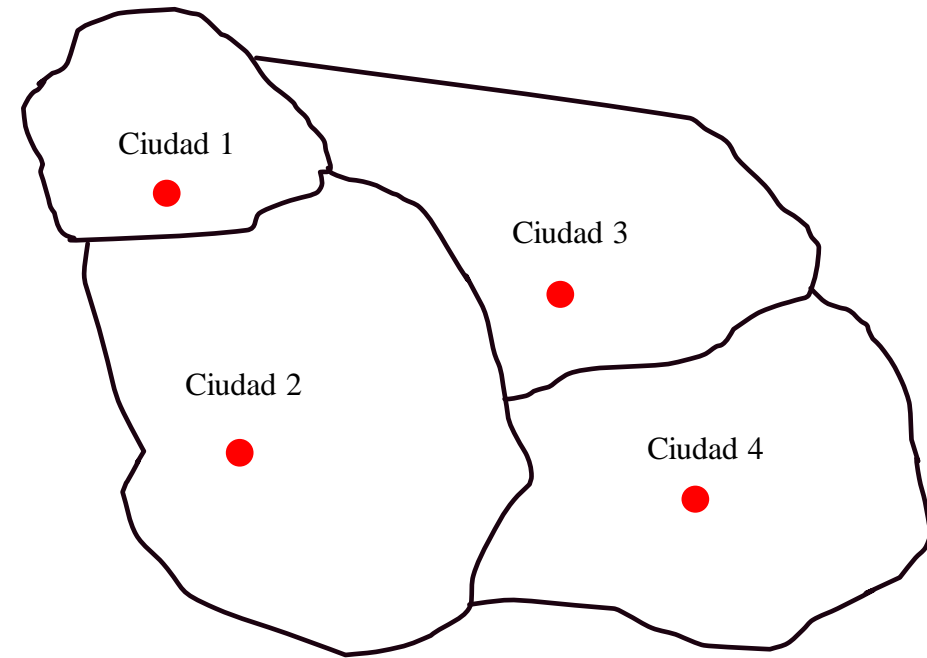
$$A = \begin{bmatrix} 0 & d_{12} & d_{13} & d_{14} \\ d_{21} & 0 & d_{23} & d_{24} \\ d_{31} & d_{32} & 0 & d_{34} \\ d_{41} & d_{42} & d_{43} & 0 \end{bmatrix}$$

Función objetivo:

$$\min f(\mathbf{x}) = d_{12}x_{12} + d_{13}x_{13} + d_{14}x_{14} + d_{21}x_{21} + d_{23}x_{23} + d_{24}x_{24} + \dots, d_{ij}x_{ij}$$

$$\min f(\mathbf{x}) = \sum_i^N \sum_j^N d_{ij}x_{ij}$$

# PROBLEMA DEL AGENTE VIAJERO: REPRESENTACIÓN EN VARIABLES BINARIAS



Matriz de costos:

$$A = \begin{bmatrix} 0 & d_{12} & d_{13} & d_{14} \\ d_{21} & 0 & d_{23} & d_{24} \\ d_{31} & d_{32} & 0 & d_{34} \\ d_{41} & d_{42} & d_{43} & 0 \end{bmatrix} = \begin{bmatrix} 0 & 10 & 2 & 1 \\ 10 & 0 & 11 & 24 \\ 2 & 11 & 0 & 35 \\ 1 & 24 & 35 & 0 \end{bmatrix}$$

Función objetivo:

$$\min f(\mathbf{x}) = \sum_i^N \sum_j^N d_{ij} x_{ij}$$

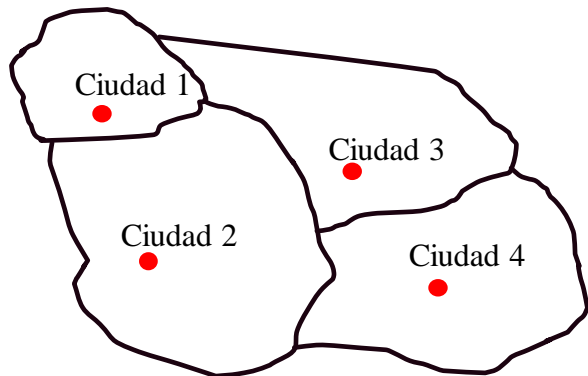
Considerando lo anterior calcule las aptitudes de los siguientes vectores:

$$\begin{aligned} \mathbf{x1} &= \begin{bmatrix} x_{1,2} & x_{1,3} & x_{1,4} & x_{2,1} & x_{2,3} & x_{2,4} & x_{3,1} & x_{3,2} & x_{3,4} & x_{4,1} & x_{4,2} & x_{4,3} \end{bmatrix} \\ \mathbf{x2} &= \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \end{aligned}$$

# PROBLEMA DEL AGENTE VIAJERO: REPRESENTACIÓN EN VARIABLES BINARIAS

$$f(\mathbf{x}) = \min \sum_i^N \sum_j^N d_{ij} x_{ij}$$

$$\text{s. a } \left. \begin{array}{l} \sum_j^N x_{ij} = 1 \text{ para } i = 1, \dots, N \\ \sum_i^N x_{ij} = 1 \text{ para } j = 1, \dots, N \end{array} \right\}$$



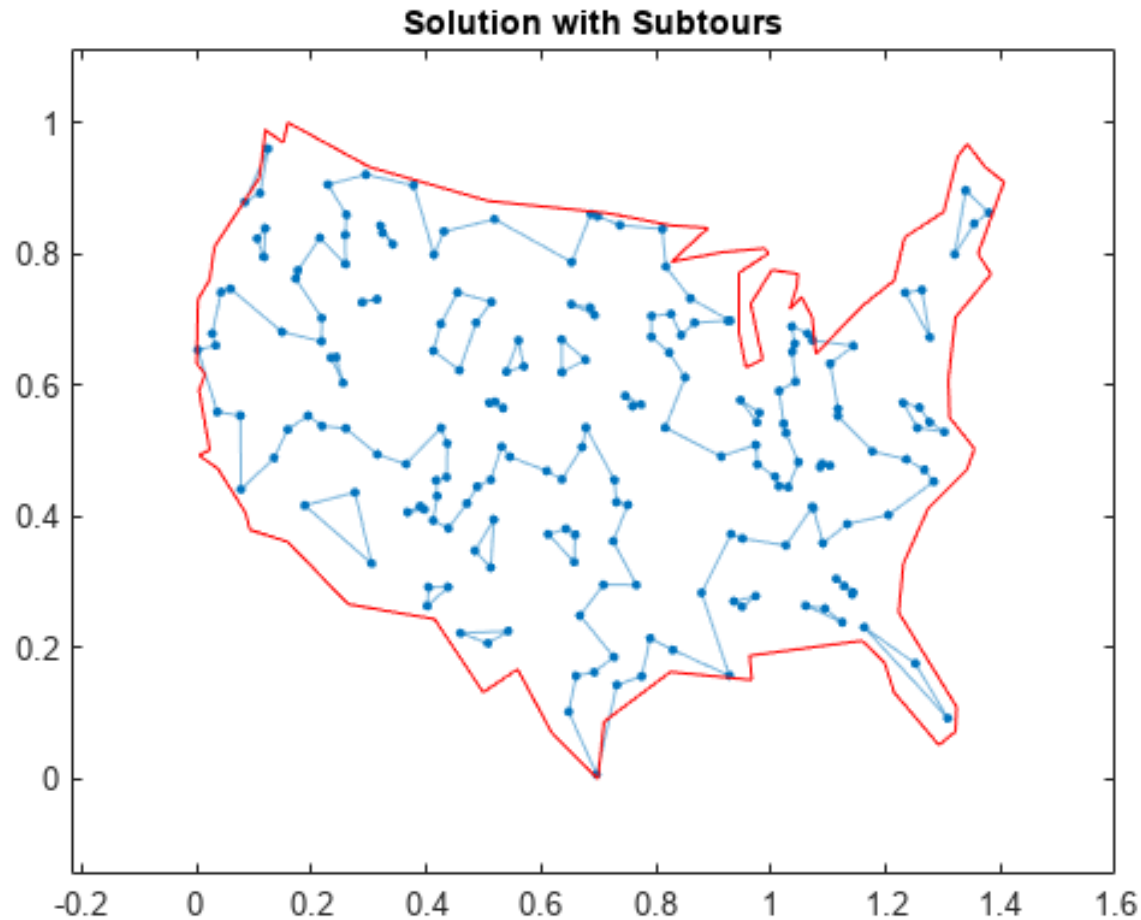
**Visitar cada ciudad exactamente una vez:** El viajante debe pasar por cada ciudad exactamente una vez antes de regresar al punto de partida. Esto garantiza que todas las ciudades sean cubiertas en la ruta.

Para el nodo 2

$$\begin{array}{ll} \sum_j^N x_{ij} = 1 \text{ para } i = 2 & x_{21} + x_{23} + x_{24} = 1 \\ \sum_i^N x_{ij} = 1 \text{ para } j = 2 & x_{12} + x_{32} + x_{42} = 1 \end{array}$$



# PROBLEMA DEL AGENTE VIAJERO: REPRESENTACIÓN EN VARIABLES BINARIAS



Un **subtour** es un ciclo que no incluye todas las ciudades. En otras palabras, si se permiten subtours en la solución, el viajante podría no visitar todas las ciudades.

# PROBLEMA DEL AGENTE VIAJERO: REPRESENTACIÓN EN VARIABLES BINARIAS

$$f(\mathbf{x}) = \min \sum_i^N \sum_j^N d_{ij} x_{ij}$$

$$s. a \sum_j^N x_{ij} = 1 \text{ para } i = 1, \dots, N$$

$$\sum_i^N x_{ij} = 1 \text{ para } j = 1, \dots, N$$

$$\sum_{i \in S}^N \sum_{j \in S}^N x_{ij} \leq |S| - 1 \text{ para } \textit{todo } S \subset V, \text{ donde } |S| > 1$$

**Restricciones de subtours:** Un subtour es un ciclo que no incluye todas las ciudades. En otras palabras, si se permiten subtours en la solución, el viajante podría no visitar todas las ciudades.

# PROBLEMA DEL AGENTE VIAJERO: REPRESENTACIÓN EN VARIABLES BINARIAS

Las dificultades surgen rápidamente cuando se aplica el simple "algoritmo genético puro" a un problema de optimización combinatoria como el TSP:

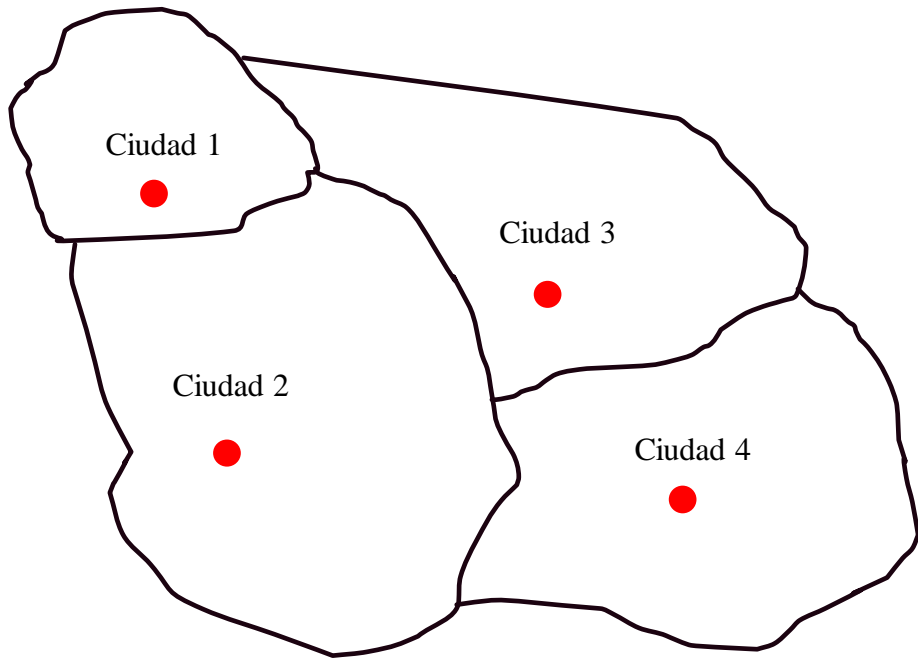
- En particular, la codificación de una solución como una cadena de bits no es conveniente. Suponiendo un TSP de tamaño  $N$ , se requieren cadenas de bits de longitud  $N(N-1)$ . En consecuencia, la mayoría de las secuencias en el espacio de búsqueda no corresponderían a recorridos factibles. Por ejemplo, sería fácil crear una secuencia con dos ocurrencias de la misma ciudad o subtours (subrutas) mediante los operadores de AG.
- En la literatura, se han propuesto funciones de aptitud con técnicas de penalización y operadores de reparación para transformar soluciones no factibles en factibles. Sin embargo, estos enfoques no siempre son relevantes en un contexto de TSP.
- El enfoque de investigación preferido para el Problema del Viajante (TSP) es diseñar marcos representacionales más sofisticados que la cadena de bits, y desarrollar operadores especializados para manipular estas representaciones y crear soluciones factibles.

# REPRESENTACIÓN POR PERMUTACIONES

**Definición:** En la representación por permutaciones, las soluciones se representan como una secuencia ordenada de elementos, donde cada elemento aparece exactamente una vez en la secuencia. Esta representación es comúnmente utilizada en problemas en los que el orden de los elementos es importante y no se pueden repetir.

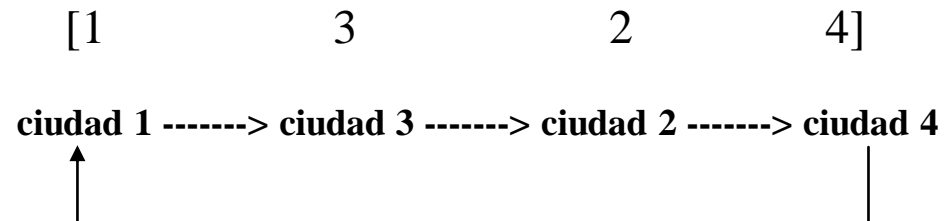
**Ejemplo:** Supongamos que tenemos 4 trabajos (A, B, C, D). Una permutación para ejecutar los trabajos podría ser [C, A, D, B], lo que indica que primero se realiza el trabajo C, luego el trabajo A, luego el trabajo D y finalmente el trabajo B.

# PROBLEMA DEL AGENTE VIAJERO: REPRESENTACIÓN POR PERMUTACIONES



En la **representación por permutación**, el vector de solución está indicando el orden en el que se conectan las ciudades en el recorrido del viajero. Cada número en el vector representa una ciudad, y la posición de ese número en el vector representa el orden en el que el viajero visita esas ciudades. **En esta representación no se repiten valores.**

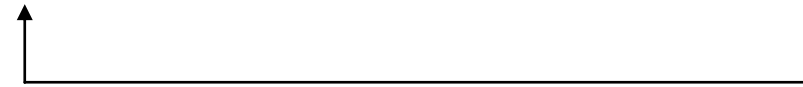
Para el problema de 4 ciudades considere que, el vector  $[1\ 3\ 2\ 4]$  representa el recorrido:



# PROBLEMA DEL AGENTE VIAJERO: REPRESENTACIÓN POR PERMUTACIÓN

[1                      3                      2                      4]

ciudad 1 -----> ciudad 3 -----> ciudad 2 -----> ciudad 4



Función objetivo

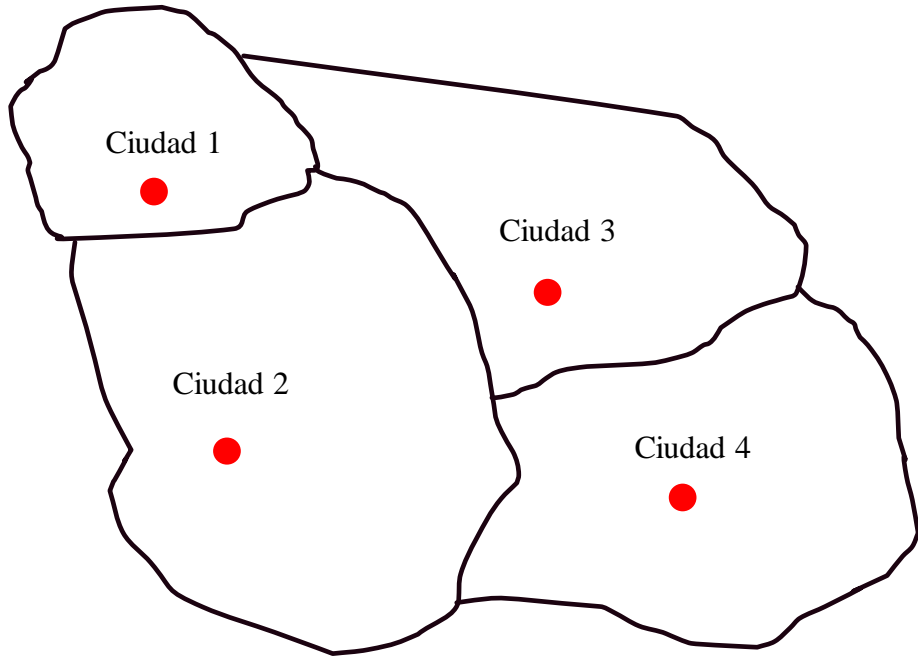
$$\min f(\mathbf{x}) = d_{x_N x_1} + \sum_{i=1}^{N-1} d_{x_i x_{i+1}}$$

s.a  $\mathbf{x} \in [1, N]$

# VENTAJAS DE REPRESENTACIÓN POR PERMUTACIÓN

- Mientras que la representación por variables binarias requiere  $N(N-1)$  variables para un problema de TSP con  $N$  ciudades, la representación por permutaciones solo necesita  $N$  variables. Esto reduce significativamente la complejidad del modelo ya que reduce la dimensionalidad.
- Garantiza que todas las ciudades sean visitadas exactamente una vez en la solución. Esto elimina la posibilidad de soluciones que representen subrutas (subtours), lo que simplifica la formulación del problema y evita la necesidad de establecer restricciones adicionales para evitar subrutas.
- Proporciona una estructura más intuitiva y comprensible, al representar el recorrido como una secuencia ordenada de ciudades.
- Se presta naturalmente a la aplicación de operadores genéticos específicos, como la recombinación de orden (OX) y la mutación de intercambio (SWAP), que son eficaces para explorar y explotar el espacio de búsqueda de soluciones.

# PROBLEMA DEL AGENTE VIAJERO: REPRESENTACIÓN POR PERMUTACIONES



1. Considerando representación por **permutación** trace las siguientes rutas:

a) [1 3 2 4]   b) [2 4 3 1]   c) [3 4 1 2]   d) [4 3 1 2]

2. Calcular las aptitudes utilizando la matriz de costos proporcionada y la función de aptitud  $f(x)$  dada:

$$\min f(\mathbf{x}) = d_{x_N x_1} + \sum_{i=1}^N d_{x_i x_{i+1}}$$

$$A = \begin{bmatrix} 0 & d_{12} & d_{13} & d_{14} \\ d_{21} & 0 & d_{23} & d_{24} \\ d_{31} & d_{32} & 0 & d_{34} \\ d_{41} & d_{42} & d_{43} & 0 \end{bmatrix} = \begin{bmatrix} 0 & 10 & 2 & 1 \\ 10 & 0 & 11 & 24 \\ 2 & 11 & 0 & 35 \\ 1 & 24 & 35 & 0 \end{bmatrix}$$



# ALGORITMO GENÉTICO HÍBRIDO (HGA)

- 1. Operador de Cruce Cycle Crossover (CX):** Este operador de cruce recombina las rutas de los padres para generar una descendencia que hereda las características favorables de ambos, evitando inconsistencias y soluciones no viables.
- 2. Heurísticas de Mejora de la Descendencia:** Las heurísticas "Remoción de Abruptos" se centra en mejorar la calidad de la descendencia generada durante el cruce. Elimina aumentos bruscos en el costo del recorrido causados por una ciudad mal posicionada.
- 3. Incorporación de Aleatoriedad:** Se introduce una cantidad apropiada de aleatoriedad mediante el operador de mezcla para evitar quedarse atrapado en óptimos locales. Esta aleatoriedad ayuda a diversificar la población y a promover la exploración de soluciones alternativas.

# PSEUDOCÓDIGO DEL ALGORITMO GENÉTICO HÍBRIDO (HGA)

## Paso 1:

- Generación aleatoria de la población inicial (generar permutaciones aleatorias).
- Evaluar la aptitud de la población inicial

## Paso 2:

- Aplicar el algoritmo "Remoción de Abruptos" a todas las rutas en la población inicial.

## Paso 3:

- Selección de padres aleatoriamente.
- Aplicar el cruce por " Cycle crossover (ER)" entre los padres y generar **un** descendiente.
- Evaluar la aptitud del descendiente.
- Aplicar la heurística "Remoción de Abruptos" al descendiente.

## Paso 4

- Ordenar los padres y el descendiente de acuerdo a su aptitud (Familia ), y pasan a conformar la siguiente generación los dos mejores individuos.



## Paso 5:

- Mezclar aleatoriamente una ruta seleccionada al azar de la población.

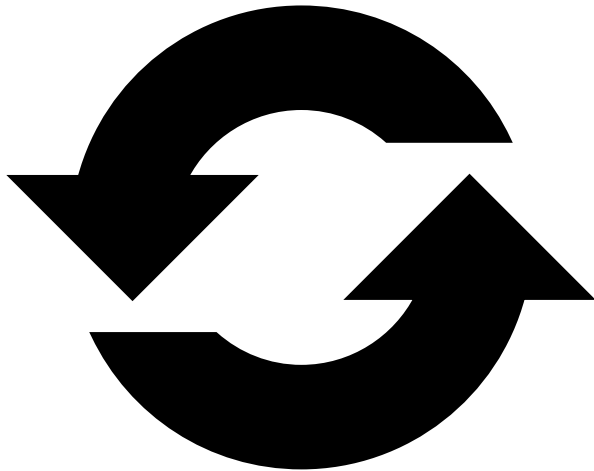
Si  $\text{rand} < \text{pm}$  (para cada individuo)  
Se genera un individuo aleatoriamente y se  
sustituye en un miembro de la población aleatorio  
end

## Paso 6:

- Repetir los pasos 3, 4, y 5 hasta el final del número especificado de generaciones.

Ciclo  
principal

# CRUZAMIENTO CYCLE CROSSOVER (CX)



El **cycle crossover (CX)** es un operador de recombinación en algoritmos genéticos que garantiza que cada descendiente herede la estructura de ambos padres sin repetir genes. El proceso comienza seleccionando una posición inicial en los padres y copiando el gen del primer padre en el primer descendiente. Luego, se busca el valor correspondiente en el segundo padre y se copia en la misma posición en el segundo descendiente. En los ciclos siguientes alternan las asignaciones. Esto asegura que los descendientes sean combinaciones válidas de ambos padres.

# CRUZAMIENTO CYCLE CROSSOVER (CX)

Padre 1=

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

Padre 2=

9	3	7	8	2	6	5	1	4
---	---	---	---	---	---	---	---	---

Hijo 1=

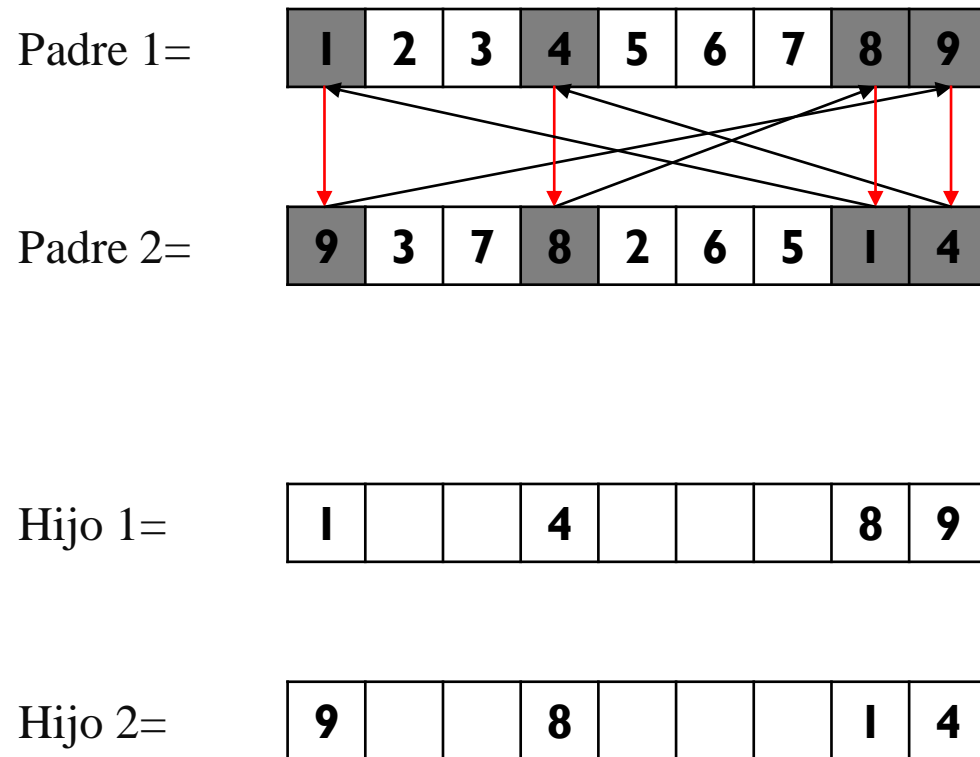
--	--	--	--	--	--	--	--	--

Hijo 2=

--	--	--	--	--	--	--	--	--

1. El valor en la primera posición del Padre 1 se coloca en la primera posición del **Hijo 1**, y el valor **aparejado** del **Padre 2** se coloca en la primera posición del **Hijo 2**.
2. A continuación, se busca en el **Padre 1** la posición donde se encuentra el valor que se acaba de colocar desde el **Padre 2**. El valor en esa nueva posición del **Padre 1** se asigna al **Hijo 1**, y el valor **aparejado** del **Padre 2** se coloca en la misma posición del **Hijo 2**.
3. Este proceso se repite hasta que se completa el ciclo, es decir, hasta que se regresa a un valor ya transmitido al hijo.

# CRUZAMIENTO CYCLE CROSSOVER (CX)



1. El valor en la primera posición del Padre 1 se coloca en la primera posición del **Hijo 1**, y el valor **aparejado** del **Padre 2** se coloca en la primera posición del **Hijo 2**.
2. A continuación, se busca en el **Padre 1** la posición donde se encuentra el valor que se acaba de colocar desde el **Padre 2**. El valor en esa nueva posición del **Padre 1** se asigna al **Hijo 1**, y el valor **aparejado** del **Padre 2** se coloca en la misma posición del **Hijo 2**.
3. Este proceso se repite hasta que se completa el ciclo, es decir, hasta que se regresa a un valor ya transmitido al hijo.

# CRUZAMIENTO CYCLE CROSSOVER (CX)

Padre 1=

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

Padre 2=

9	3	7	8	2	6	5	1	4
---	---	---	---	---	---	---	---	---

Hijo 1=

1			4				8	9
---	--	--	---	--	--	--	---	---

Hijo 2=

9			8				1	4
---	--	--	---	--	--	--	---	---

1. El segundo ciclo comienza en la primera posición del **Padre 2** no transmitida.
2. Ahora, se invierte el orden de transmisión, el **Padre 1** transmite al **Hijo 2** y el **Padre 2** transmite al **Hijo 1**.
3. Este proceso se realiza hasta concluir el ciclo. Con el inicio de cada ciclo el orden de transmisión se invierte.

# CRUZAMIENTO CYCLE CROSSOVER (CX)

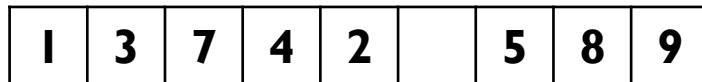
Padre 1=



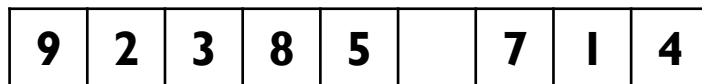
Padre 2=



Hijo 1=



Hijo 2=



1. El segundo ciclo comienza en la primera posición del **Padre 2** no transmitida.
2. Ahora, se invierte el orden de transmisión, el **Padre 1** transmite al **Hijo 2** y el **Padre 2** transmite al **Hijo 1**.
3. Este proceso se realiza hasta concluir el ciclo. Con el inicio de cada ciclo el orden de transmisión se invierte.

# CRUZAMIENTO CYCLE CROSSOVER (CX)

Padre 1= 

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

Padre 2= 

9	3	7	8	2	6	5	1	4
---	---	---	---	---	---	---	---	---

Hijo 1= 

1	3	7	4	2	6	5	8	9
---	---	---	---	---	---	---	---	---

Hijo 2= 

9	2	3	8	5	6	7	1	4
---	---	---	---	---	---	---	---	---

1. En el tercer ciclo se busca el primer elemento del Padre 1 que no se ha transmitido y se vuelve a invertir el sentido de la transmisión. Ahora padre 1 transmite a Hijo 1, y Padre 2 a Hijo 2.
2. Cuando todos los elementos fueron transmitidos termina la ejecución del cruzamiento



# CRUZAMIENTO CYCLE CROSSOVER (CX)

Padre 1=

4	9	1	3	5	6	7	8	2
---	---	---	---	---	---	---	---	---

Padre 2=

3	7	4	2	9	6	5	1	8
---	---	---	---	---	---	---	---	---

Hijo 1=

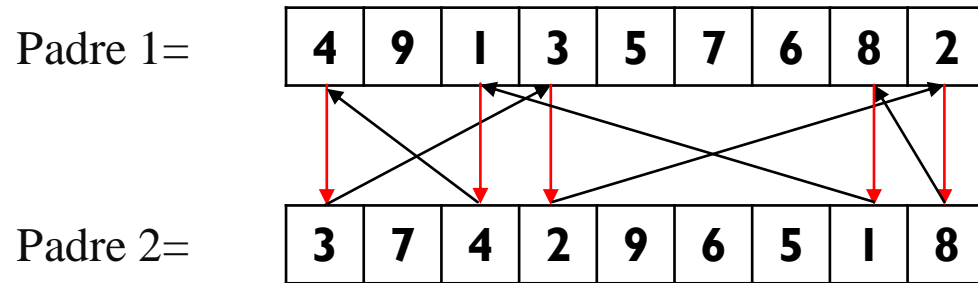
--	--	--	--	--	--	--	--	--

Hijo 2=

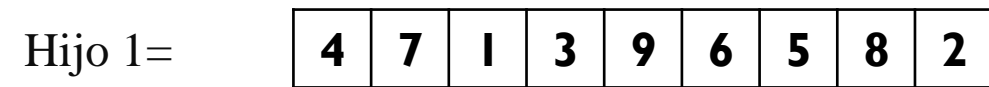
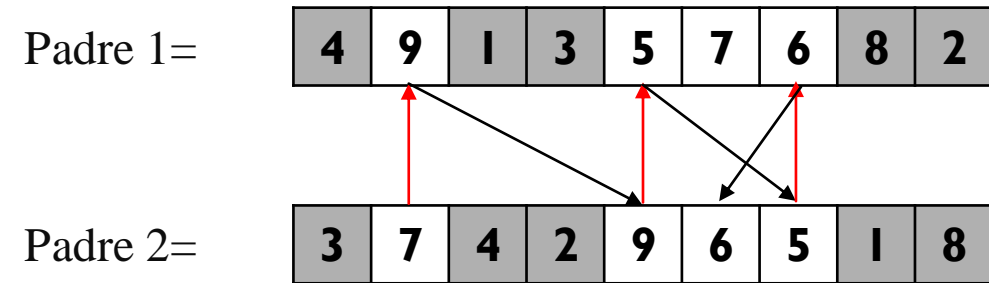
--	--	--	--	--	--	--	--	--

# CRUZAMIENTO CYCLE CROSSOVER (CX)

Ciclo 1



Ciclo 2



# CRUZAMIENTO CYCLE CROSSOVER (PSEUDOCÓDIGO)

```
%Iniciar ciclo en cero

%Iniciar posición 1 y valores de la posición 1

%Iniciar un registro de las posiciones visitadas

while true

    % Asignar valor 1 valor 2 a hijo 1 e hijo 2

    % Registrar el valor ya visitado

    % Hallar la nueva posición

    %Verificar si todos los valores fueron visitados(break)

    %Verificar si la nueva posición ya fue visitada(cambio de ciclo)

        %Alternar ciclo
        si ciclo=0 entonces ciclo=1
        si ciclo=1 entonces ciclo=0

        %Seleccionar la posición del próximo ciclo

    % determinar valor 1 y valor 2(dependiendo del sentido del ciclo)
```

# CRUZAMIENTO EDGE RECOMBINATION (ER)


Paso 1:

- Crear lista de ciudades vecinas

Padre 1: [1 5 3 2 4]

Padre 2: [4 3 1 5 2]

Ciudades	Ciudades vecinas
1	4 5 3
2	
3	
4	
5	

- 
- En el padre 1, la ciudad 1 tiene de vecino (al lado) a la ciudad 5, también tiene de vecino a 4 porque de 4 se vuelve a 1.
  - En el padre 2, la ciudad 1 tiene de vecino a 3 y a 5

# CRUZAMIENTO EDGE RECOMBINATION (ER)

Paso 2:

- Para la primera posición del hijo se selecciona aleatoriamente la ciudad inicial de uno de los padres. En este caso se selecciona aleatoriamente entre 1 y 4. Esta selección toma la condición de "ciudad actual".

Padre 1: [1 5 3 2 4]

Padre 2: [4 3 1 5 2]

Hijo= 

4				
---	--	--	--	--

Ciudad actual →

Ciudades	Ciudades vecinas
1	4 5 3 2
2	3 4 5
3	5 2 4 1
4	2 1 3
5	1 3 2

# CRUZAMIENTO EDGE RECOMBINATION (ER)

Paso 3:

**WHILE** el hijo no esté completo



- Eliminar todas las veces que aparezca la "ciudad actual" en las ciudades vecinas.

**IF** la "ciudad actual" aún tiene ciudades vecinas:

- Determinar cuál de las ciudades vecinas de la "ciudad actual" tiene menos vecinos. Seleccionar esa ciudad para incluirla en la ruta y convertirla en la "ciudad actual". En caso de empate, resolver aleatoriamente.

**ELSE**

- Elegir aleatoriamente una ciudad no visitada y convertirla en "ciudad actual".

**END**

**END**

Ciudades	Ciudades vecinas
1	<del>4</del> 5 3 2
2	3 <del>4</del> 5
3	5 2 <del>4</del> 1
4	2 1 3
5	1 3 2

Hijo=

4				
---	--	--	--	--

# CRUZAMIENTO EDGE RECOMBINATION (ER)

Paso 3:

**WHILE** el hijo no esté completo



- Eliminar todas las veces que aparezca la "ciudad actual" en las ciudades vecinas.

**IF** la "ciudad actual" aún tiene ciudades vecinas:

- Determinar cuál de las ciudades vecinas de la "ciudad actual" tiene menos vecinos. Seleccionar esa ciudad para incluirla en la ruta y convertirla en la "ciudad actual". En caso de empate, resolver aleatoriamente.

**ELSE**

- Elegir aleatoriamente una ciudad no visitada y convertirla en "ciudad actual".

**END**

**END**

Ciudades	Ciudades vecinas
1	5 3 2
2	3 5
3	5 2 1
4	2 1 3
5	1 3 2

Hijo=

4				
---	--	--	--	--


# CRUZAMIENTO EDGE RECOMBINATION (ER)

Paso 3:

**WHILE** el hijo no esté completo

- Eliminar todas las veces que aparezca la "ciudad actual" en las ciudades vecinas.

**IF** la "ciudad actual" aún tiene ciudades vecinas:

- 
- Determinar cuál de las ciudades vecinas de la "ciudad actual" tiene menos vecinos. Seleccionar esa ciudad para incluirla en la ruta y convertirla en la "ciudad actual". En caso de empate, resolver aleatoriamente.

**ELSE**

- Elegir aleatoriamente una ciudad no visitada y convertirla en "ciudad actual".

**END**

**END**

Ciudades	Ciudades vecinas
1	5 3 2
2	3 5
3	5 2 1
4	2 1 3
5	1 3 2

Hijo=

4				
---	--	--	--	--




# CRUZAMIENTO EDGE RECOMBINATION (ER)

Paso 3:

**WHILE** el hijo no esté completo

- Eliminar todas las veces que aparezca la "ciudad actual" en las ciudades vecinas.

**IF** la "ciudad actual" aún tiene ciudades vecinas:

- 
- Determinar cuál de las ciudades vecinas de la "ciudad actual" tiene menos vecinos. Seleccionar esa ciudad para incluirla en la ruta y convertirla en la "ciudad actual". En caso de empate, resolver aleatoriamente.

**ELSE**

- Elegir aleatoriamente una ciudad no visitada y convertirla en "ciudad actual".

**END**

**END**

Ciudades	Ciudades vecinas
1	5 3 2
2	3 5
3	5 2 1
4	2 1 3
5	1 3 2


Hijo= 

4	2			
---	---	--	--	--

# CRUZAMIENTO EDGE RECOMBINATION (ER)

Paso 3:

**WHILE** el hijo no esté completo

- 
- Eliminar todas las veces que aparezca la "ciudad actual" en las ciudades vecinas.

**IF** la "ciudad actual" aún tiene ciudades vecinas:

- Determinar cuál de las ciudades vecinas de la "ciudad actual" tiene menos vecinos. Seleccionar esa ciudad para incluirla en la ruta y convertirla en la "ciudad actual". En caso de empate, resolver aleatoriamente.

**ELSE**

- Elegir aleatoriamente una ciudad no visitada y convertirla en "ciudad actual".

**END**

**END**

Ciudades	Ciudades vecinas
1	5 3 <del>2</del>
2	3 5
3	5 <del>2</del> 1
4	<del>2</del> 1 3
5	1 3 <del>2</del>

Hijo= 

4	2			
---	---	--	--	--


# CRUZAMIENTO EDGE RECOMBINATION (ER)

Paso 3:

**WHILE** el hijo no esté completo

- Eliminar todas las ocurrencias de la "ciudad actual" del lado izquierdo del mapa de bordes.

**IF** la "ciudad actual" aún tiene ciudades vecinas:

- 
- Determinar cuál de las ciudades vecinas de la "ciudad actual" tiene menos vecinos. Seleccionar esa ciudad para incluirla en la ruta y convertirla en la "ciudad actual". **En caso de empate, resolver aleatoriamente.**

**ELSE**

- Elegir aleatoriamente una ciudad no visitada y convertirla en "ciudad actual".

**END**

**END**

Ciudades	Ciudades vecinas
1	5 3
2	3 5
3	5 1
4	1 3
5	1 3

Hijo= 

4	2			
---	---	--	--	--


# CRUZAMIENTO EDGE RECOMBINATION (ER)

Paso 3:

**WHILE** el hijo no esté completo

- Eliminar todas las veces que aparezca la "ciudad actual" en las ciudades vecinas.

**IF** la "ciudad actual" aún tiene ciudades vecinas:

- 
- Determinar cuál de las ciudades vecinas de la "ciudad actual" tiene menos vecinos. Seleccionar esa ciudad para incluirla en la ruta y convertirla en la "ciudad actual". **En caso de empate, resolver aleatoriamente.**

**ELSE**

- Elegir aleatoriamente una ciudad no visitada y convertirla en "ciudad actual".

**END**

**END**

Ciudades	Ciudades vecinas
1	5 3
2	3 5
3	5 1
4	1 3
5	1 3

Hijo= 

4	2	3		
---	---	---	--	--

# CRUZAMIENTO EDGE RECOMBINATION (ER)

Paso 3:

**WHILE** el hijo no esté completo

- Eliminar todas las veces que aparezca la "ciudad actual" en las ciudades vecinas.

**IF** la "ciudad actual" aún tiene ciudades vecinas:

- Determinar cuál de las ciudades vecinas de la "ciudad actual" tiene menos vecinos. Seleccionar esa ciudad para incluirla en la ruta y convertirla en la "ciudad actual". En caso de empate, resolver aleatoriamente.

**ELSE**

- Elegir aleatoriamente una ciudad no visitada y convertirla en "ciudad actual".

**END**

**END**

Ciudades	Ciudades vecinas
1	5 <del>3</del>
2	<del>3</del> 5
3	5 1
4	1 <del>3</del>
5	1 <del>3</del>

Hijo= 

4	2	3		
---	---	---	--	--


# CRUZAMIENTO EDGE RECOMBINATION (ER)

Paso 3:

**WHILE** el hijo no esté completo

- Eliminar todas las veces que aparezca la "ciudad actual" en las ciudades vecinas.

**IF** la "ciudad actual" aún tiene ciudades vecinas:

- 
- Determinar cuál de las ciudades vecinas de la "ciudad actual" tiene menos vecinos. Seleccionar esa ciudad para incluirla en la ruta y convertirla en la "ciudad actual". **En caso de empate, resolver aleatoriamente.**

**ELSE**

- Elegir aleatoriamente una ciudad no visitada y convertirla en "ciudad actual".

**END**

**END**

Ciudades	Ciudades vecinas
1	5
2	5
3	5 1
4	1
5	1


Hijo= 

4	2	3		
---	---	---	--	--

# CRUZAMIENTO EDGE RECOMBINATION (ER)

Paso 3:

**WHILE** el hijo no esté completo

- 
- Eliminar todas las veces que aparezca la "ciudad actual" en las ciudades vecinas.

**IF** la "ciudad actual" aún tiene ciudades vecinas:

- Determinar cuál de las ciudades vecinas de la "ciudad actual" tiene menos vecinos. Seleccionar esa ciudad para incluirla en la ruta y convertirla en la "ciudad actual". En caso de empate, resolver aleatoriamente.

**ELSE**

- Elegir aleatoriamente una ciudad no visitada y convertirla en "ciudad actual".

**END**

**END**

Ciudades	Ciudades vecinas
1	<del>5</del>
2	<del>5</del>
3	<del>5</del> 1
4	1
5	1

Hijo= 

4	2	3	5	
---	---	---	---	--


# CRUZAMIENTO EDGE RECOMBINATION (ER)

Paso 3:

**WHILE** el hijo no esté completo

- Eliminar todas las veces que aparezca la "ciudad actual" en las ciudades vecinas.

**IF** la "ciudad actual" aún tiene ciudades vecinas:

- 
- Determinar cuál de las ciudades vecinas de la "ciudad actual" tiene menos vecinos. Seleccionar esa ciudad para incluirla en la ruta y convertirla en la "ciudad actual". En caso de empate, resolver aleatoriamente.

**ELSE**

- Elegir aleatoriamente una ciudad no visitada y convertirla en "ciudad actual".

**END**

**END**

Ciudades	Ciudades vecinas
1	
2	
3	1
4	1
5	1

Hijo= 

4	2	3	5	
---	---	---	---	--



# CRUZAMIENTO EDGE RECOMBINATION (ER)

Paso 3:

**WHILE** el hijo no esté completo

- Eliminar todas las veces que aparezca la "ciudad actual" en las ciudades vecinas.

**IF** la "ciudad actual" aún tiene ciudades vecinas:

- Determinar cuál de las ciudades vecinas de la "ciudad actual" tiene menos vecinos. Seleccionar esa ciudad para incluirla en la ruta y convertirla en la "ciudad actual". En caso de empate, resolver aleatoriamente.

**ELSE**

- Elegir aleatoriamente una ciudad no visitada y convertirla en "ciudad actual".

**END**  
**END**

Ciudades	Ciudades vecinas
1	
2	
3	1
4	1
5	1

Hijo=

4	2	3	5	1
---	---	---	---	---

# CRUZAMIENTO EDGE RECOMBINATION (ER)

- El operador de recombinación de bordes (ER) es un operador que crea una ruta similar a un conjunto de rutas existentes (padres) al observar las conexiones existentes en lugar de solamente nodos o segmentos de varias conexiones. Esto puede resultar en soluciones descendientes que preservan las buenas características de los padres. Fue descrito por Darrell Whitley en 1989.
- El algoritmo de recombinación de bordes garantiza que las soluciones descendientes sigan siendo factibles para el problema en cuestión.
- Al seleccionar el nodo con un conjunto más pequeño de ciudades vecinas, se elige una ruta que tiende a no ser explorada por los padres. Esto aumenta la diversidad en la población de soluciones descendientes, lo que es crucial para explorar ampliamente el espacio de búsqueda y evitar la convergencia prematura hacia óptimos locales.
- Cuando una ciudad ya no tiene vecinos disponibles o cuando las ciudades vecinas tienen conjuntos de igual tamaño, la introducción de la aleatoriedad garantiza que el algoritmo pueda explorar diferentes opciones y evitar estancarse en soluciones subóptimas.

# PSEUDICÓDIGO (ER)

- Crear lista de ciudades vecinas
- Para la primera posición del hijo se selecciona aleatoriamente la ciudad inicial de uno de los padres.

**WHILE** el hijo no esté completo

- Eliminar todas las veces que aparezca la "ciudad actual" en las ciudades vecinas.

**IF** la "ciudad actual" aún tiene ciudades vecinas:

- Determinar cuál de las ciudades vecinas de la "ciudad actual" tiene menos vecinos. Seleccionar esa ciudad para incluirla en la ruta y convertirla en la "ciudad actual". En caso de empate, resolver aleatoriamente.

**ELSE**

- Elegir aleatoriamente una ciudad no visitada y convertirla en "ciudad actual".

**END**

**END**

Calcular hijo considerando:

Padre 1: [2 4 1 5 3]

Padre 2: [3 4 5 1 2]

Ciudades	Ciudades vecinas
1	
2	
3	
4	
5	

Hijo=

--	--	--	--	--

# CRUZAMIENTO EDGE RECOMBINATION (ER)

Calcular:

Padre 1: [2 4 1 5 3]

Padre 2: [3 4 5 1 2]

--	--	--	--	--

Ciudades	Ciudades vecinas
1	
2	
3	
4	
5	

# HEURÍSTICA: REMOCIÓN DE ABRUPTOS

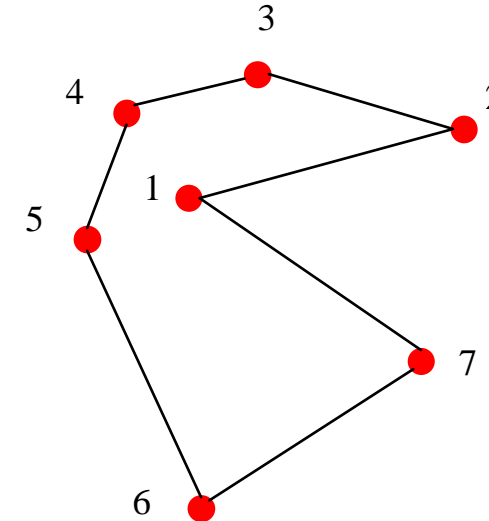
El algoritmo "Remoción de Abruptos" elimina el aumento brusco en el costo del recorrido debido a una ciudad mal posicionada. El algoritmo funciona de la siguiente manera:

Paso 1: Se crea una lista (NEARLIST) que contiene las  $m$  ciudades más cercanas a una ciudad actual.

Paso 2: Se saca la ciudad actual de su posición y se inserta antes y después de cualquiera de las ciudades del NEARLIST y se calcula el costo de la nueva longitud del recorrido para cada caso.

Paso 4: Si alguna de las nuevas rutas mejora la ruta original, la nueva ruta sustituye a la ruta original en la población.

Paso 5: Los pasos anteriores se repiten para cada ciudad en el recorrido.



# HEURÍSTICA: REMOCIÓN DE ABRUPTOS

El algoritmo "Remoción de Abruptos" elimina el aumento brusco en el costo del recorrido debido a una ciudad mal posicionada. El algoritmo funciona de la siguiente manera:

Paso 1: Se crea una lista (NEARLIST) que contiene las  $m$  ciudades más cercanas a una ciudad actual.

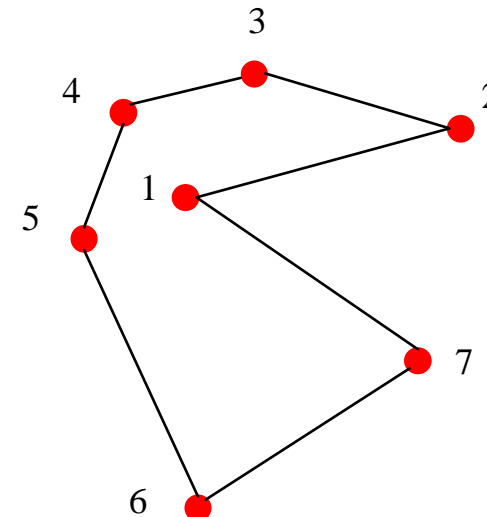
Paso 2: Se saca la ciudad actual de su posición y se inserta antes y después de cualquiera de las ciudades del NEARLIST y se calcula el costo de la nueva longitud del recorrido para cada caso.

Paso 4: Si alguna de las nuevas rutas mejora la ruta original, la nueva ruta sustituye a la ruta original en la población.

Paso 5: Los pasos anteriores se repiten para cada ciudad en el recorrido.

Estableciendo parámetro  $m=3$

- Comenzamos por ciudad 1
- Las tres ciudades más cercanas a 1 son ciudad 5, ciudad 4 y ciudad 3



# HEURÍSTICA: REMOCIÓN DE ABRUPTOS

El algoritmo "Remoción de Abruptos" elimina el aumento brusco en el costo del recorrido debido a una ciudad mal posicionada. El algoritmo funciona de la siguiente manera:

Paso 1: Se crea una lista (NEARLIST) que contiene las  $m$  ciudades más cercanas a una ciudad actual.

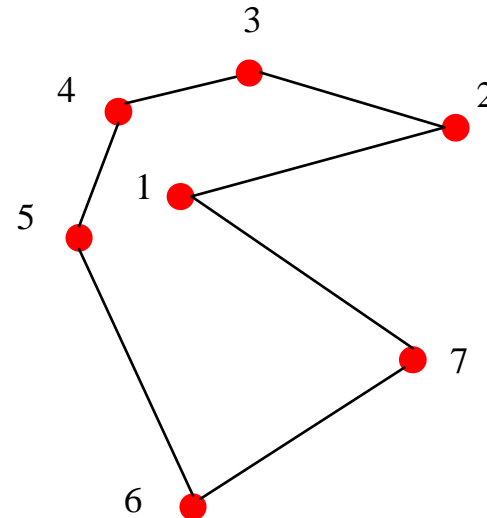
Paso 2: Se saca la ciudad actual de su posición y se inserta antes y después de cualquiera de las ciudades del NEARLIST y se calcula el costo de la nueva longitud del recorrido para cada caso.

Paso 4: Si alguna de las nuevas rutas mejora la ruta original, la nueva ruta sustituye a la ruta original en la población.

Paso 5: Los pasos anteriores se repiten para cada ciudad en el recorrido.

Estableciendo parámetro  $m=3$

- Comenzamos por ciudad 1
- Seleccionamos aleatoriamente a una de las ciudades cercanas (supongamos ciudad 5)



# HEURÍSTICA: REMOCIÓN DE ABRUPTOS

El algoritmo "Remoción de Abruptos" elimina el aumento brusco en el costo del recorrido debido a una ciudad mal posicionada. El algoritmo funciona de la siguiente manera:

Paso 1: Se crea una lista (NEARLIST) que contiene las  $m$  ciudades más cercanas a una ciudad actual.

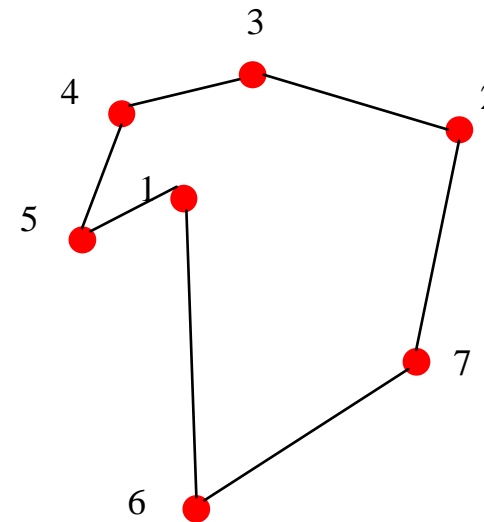
Paso 2: Se saca la ciudad actual de su posición y se inserta antes y después de cualquiera de las ciudades del NEARLIST y se calcula el costo de la nueva longitud del recorrido para cada caso.

Paso 4: Si alguna de las nuevas rutas mejora la ruta original, la nueva ruta sustituye a la ruta original en la población.

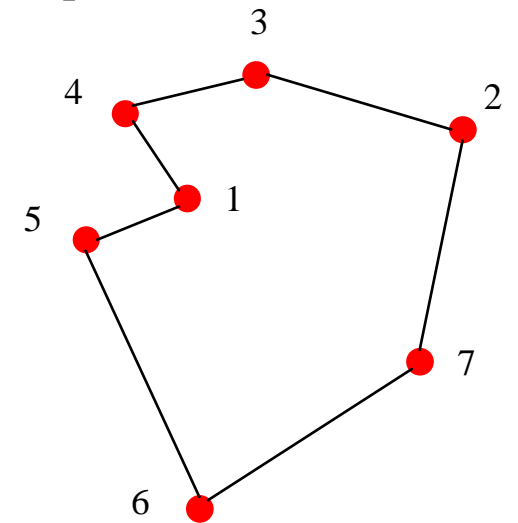
Paso 5: Los pasos anteriores se repiten para cada ciudad en el recorrido.

Estableciendo parámetro  $m=3$

- Comenzamos por ciudad 1
- Insertamos la ciudad 1 antes y después de la ciudad 5



[6 5 4...2 1 7]  
↑  
└───┘



[6 5 4...2 1 7]  
↑  
└───┘



# HEURÍSTICA: REMOCIÓN DE ABRUPTOS

El algoritmo "Remoción de Abruptos" elimina el aumento brusco en el costo del recorrido debido a una ciudad mal posicionada. El algoritmo funciona de la siguiente manera:

Paso 1: Se crea una lista (NEARLIST) que contiene las  $m$  ciudades más cercanas a una ciudad actual.

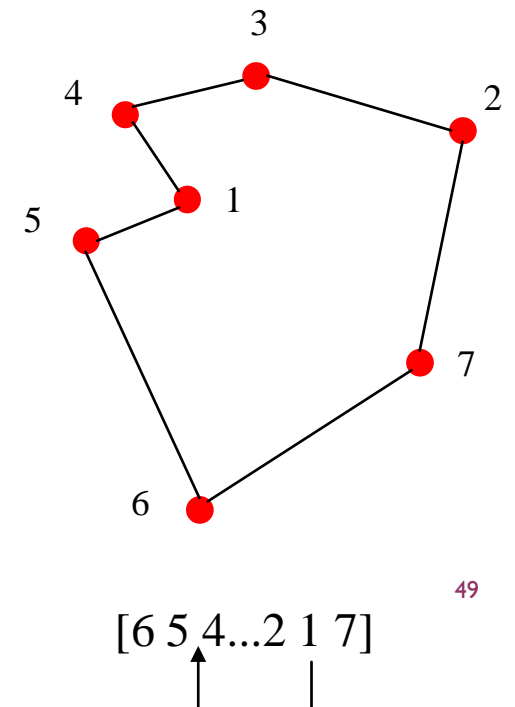
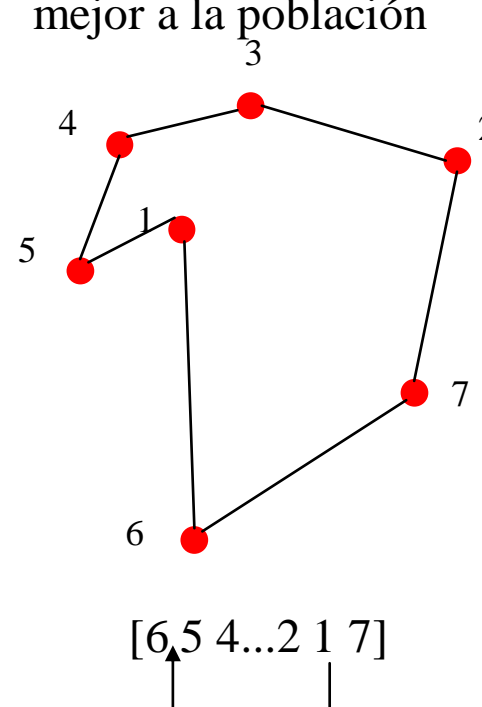
Paso 2: Se saca la ciudad actual de su posición y se inserta antes y después de cualquiera de las ciudades del NEARLIST y se calcula el costo de la nueva longitud del recorrido para cada caso.

Paso 4: Si alguna de las nuevas rutas mejora la ruta original, la nueva ruta sustituye a la ruta original en la población.

Paso 5: Los pasos anteriores se repiten para cada ciudad en el recorrido.

Estableciendo parámetro  $m=3$

- Comenzamos por ciudad 1
- Comparamos las dos rutas con la ruta original y pasa la mejor a la población



# HEURÍSTICA: REMOCIÓN DE ABRUPTOS

El algoritmo "Remoción de Abruptos" elimina el aumento brusco en el costo del recorrido debido a una ciudad mal posicionada. El algoritmo funciona de la siguiente manera:

Paso 1: Se crea una lista (NEARLIST) que contiene las ***m*** ciudades más cercanas a una ciudad actual.

Paso 2: Se saca la ciudad actual de su posición y se inserta antes y después de cualquiera de las ciudades del NEARLIST y se calcula el costo de la nueva longitud del recorrido para cada caso.

Paso 4: Si alguna de las nuevas rutas mejora la ruta original, la nueva ruta sustituye a la ruta original en la población.

Paso 5: Los pasos anteriores se repiten para cada ciudad en el recorrido.

Estableciendo parámetro  $m=3$

- Comenzamos por ciudad 1
- El proceso se repite para cada ciudad

# PSEUDICÓDIGO (ER)

- Crear lista de ciudades vecinas
- Para la primera posición del hijo se selecciona aleatoriamente la ciudad inicial de uno de los padres.

**WHILE** el hijo no esté completo

- Eliminar todas las veces que aparezca la "ciudad actual" en las ciudades vecinas.

**IF** la "ciudad actual" aún tiene ciudades vecinas:

- Determinar cuál de las ciudades vecinas de la "ciudad actual" tiene menos vecinos. Seleccionar esa ciudad para incluirla en la ruta y convertirla en la "ciudad actual". En caso de empate, resolver aleatoriamente.

**ELSE**

- Elegir aleatoriamente una ciudad no visitada y convertirla en "ciudad actual".

**END**

**END**

# CÓDIGO (ER)

- Crear lista de ciudades vecinas

```
function ListaCiudadesVecinas = CreadorLista(Padre1, Padre2, numCiudades)
```

```
%arreglo de celdas
```

```
ListaCiudadesVecinas = cell(1, numCiudades);
```

```
for i = 1:numCiudades
```

```
    Ciudad = Padre1(i);
```

```
    idx1 = [i-1, i+1];
```

```
    idx2 = find(Padre2 == Ciudad);
```

```
    idx2 = [idx2-1, idx2+1];
```

```
% Manejar los casos en los que idx1 o idx2 están fuera de los límites
```

```
    idx1(idx1 == 0) = numCiudades;
```

```
    idx1(idx1 > numCiudades) = 1;
```

```
    idx2(idx2 == 0) = numCiudades;
```

```
    idx2(idx2 > numCiudades) = 1;
```

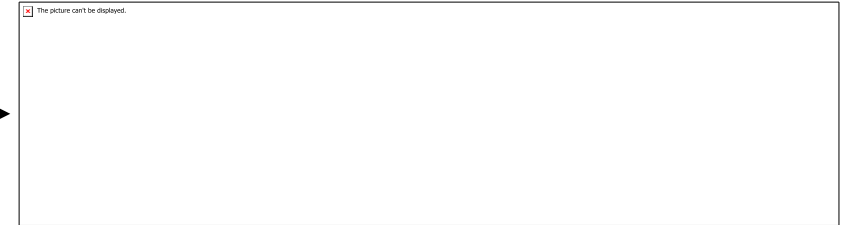
```
%Ciudades vecinas no repetidas
```

```
    vecinas = unique([Padre1(idx1), Padre2(idx2)]);
```

```
    ListaCiudadesVecinas{Ciudad} = vecinas;
```

```
end
```

```
end
```



Calcular:

Padre 1: [2 5 3 1 4]

Padre 2: [4 3 1 5 2]



# CÓDIGO (ER)

Calcular:

Padre 1: [2 5 3 1 4]

Padre 2: [4 3 1 5 2]

```
function Hijo = edgeRecombination(Padre1, Padre2, numCiudades)
```

```
% Inicialización
```

```
Hijo = zeros(1, numCiudades);
```

```
% Paso 1: Crear la lista de ciudades vecinas
```

```
ListaCiudadesVecinas = CreadorLista(Padre1, Padre2, numCiudades);
```

```
% Paso 2: Seleccionar la primera ciudad
```

```
rd=round(rand);
```

```
CiudadActual= rd*Padre1(1)+(1-rd)*Padre2(1);
```

```
Hijo(1) = CiudadActual;
```

```
% Paso 3: Recorrer las ciudades restantes
```

```
for i = 2:numCiudades
```

```
% Actualizar la lista de ciudades vecinas
```

```
for j=1:length(ListaCiudadesVecinas)
```

```
idx=ListaCiudadesVecinas{j}==CiudadActual;
```

```
ListaCiudadesVecinas{j}(idx)=[];
```

```
end
```

```
% Ciudades vecinas a la CiudadActual
```

```
VecindarioActual =ListaCiudadesVecinas{CiudadActual};
```

[4 5]

```
if isempty(VecindarioActual);
```

```
% Determinar las ciudades no visitadas y elegir una aleatoria
```

```
allCities = setdiff(1:numCiudades, Hijo(1:i-1));
```

```
CiudadActual = randsample(allCities, 1);
```

```
else
```

```
%Num de conexiones de ciudades vecinas
```

```
Nconex=[];
```

```
for j=1:length(VecindarioActual)
```

```
Nconex(end+1)=length(ListaCiudadesVecinas{VecindarioActual(j)}); longitud del vecindario de 4 y de 5 (miembros de VecindarioActual)
```

```
end
```

```
% Encuentra los índices de los valores mínimos
```

```
[~, Minidx] = find(Nconex == min(Nconex));
```

4 y 5 tienen conjuntos de 2 ciudades

```
% Selecciona aleatoriamente uno de los índices mínimos
```

```
CiudadActual = VecindarioActual(Minidx(randi(length(Minidx)))); se elige entre 4 y 5 aleatoriamente
```

```
end
```

```
Hijo(i) = CiudadActual;
```

```
end
```

```
end
```

} Selección aleatoria entre 2 y 4 y poner en primera posición del hijo

} posición en ListaCiudadesVecinas{1,2,...,5}==2  
borrar elementos en la posición

	1	2	3	4	5
1	[3,4,5]	[4,5]	[1,4,5]	[1,3]	[1,3]
2					
3					

# CÓDIGO (ER)

Calcular:

Padre 1: [2 5 3 1 4]

Padre 2: [4 3 1 5 2]

```
function Hijo = edgeRecombination(Padre1, Padre2, numCiudades)
```

```
% Inicialización
```

```
Hijo = zeros(1, numCiudades);
```

```
% Paso 1: Crear la lista de ciudades vecinas
```

```
ListaCiudadesVecinas = CreadorLista(Padre1, Padre2, numCiudades);
```

```
% Paso 2: Seleccionar la primera ciudad
```

```
rd=round(rand);
```

```
CiudadActual= rd*Padre1(1)+(1-rd)*Padre2(1);
```

```
Hijo(1) = CiudadActual;
```

```
% Paso 3: Recorrer las ciudades restantes
```

```
for i = 2:numCiudades
```

```
% Actualizar la lista de ciudades vecinas
```

```
for j=1:length(ListaCiudadesVecinas)
```

```
idx=ListaCiudadesVecinas{j}==CiudadActual;
```

```
ListaCiudadesVecinas{j}(idx)=[];
```

```
end
```

```
% Ciudades vecinas a la CiudadActual
```

```
VecindarioActual =ListaCiudadesVecinas{CiudadActual};
```

[4 5]

```
if isempty(VecindarioActual);
```

```
% Determinar las ciudades no visitadas y elegir una aleatoria
```

```
allCities = setdiff(1:numCiudades, Hijo(1:i-1));
```

```
CiudadActual = randsample(allCities, 1);
```

```
else
```

```
%Num de conexiones de ciudades vecinas
```

```
Nconex=[];
```

```
for j=1:length(VecindarioActual)
```

```
Nconex(end+1)=length(ListaCiudadesVecinas{VecindarioActual(j)}); longitud del vecindario de 4 y de 5 (miembros de VecindarioActual)
```

```
end
```

```
% Encuentra los índices de los valores mínimos
```

```
[~, Minidx] = find(Nconex == min(Nconex));
```

4 y 5 tienen conjuntos de 2 ciudades

```
% Selecciona aleatoriamente uno de los índices mínimos
```

```
CiudadActual = VecindarioActual(Minidx(randi(length(Minidx)))); se elige entre 4 y 5 aleatoriamente
```

```
end
```

```
Hijo(i) = CiudadActual;
```

```
end
```

```
end
```

} Selección aleatoria entre 2 y 4 y poner en primera posición del hijo

} posición en ListaCiudadesVecinas{1,2,...,5}==2  
borrar elementos en la posición

	1	2	3	4	5
1	[3,4,5]	[4,5]	[1,4,5]	[1,3]	[1,3]
2					
3					

# Heurística: Remoción de Abruptos

```
%Hijo
Hijo=[4 3 1 5 2];
%parámetro ciudades cercanas
m=3;
%matriz de distancias
d=[ 0 10 5 3 2; 10 0 3 4 1; 5 3 0 13 8; 3 4 13 0 7; 2 1 8 7 0];
%número de ciudades
numCiudades = length(Hijo);
```

```
for i=1:numCiudades
```

```
%Selección entre las ciudades más cercanas
```

```
[~,idx]=sort(d(i,:));
```

```
idx=idx(2:m+1);
```

```
idx=randsample(idx,1);
```

```
%Posición de inserción
```

```
posiciones =find(Hijo==idx);
```

```
posiciones =[posiciones posiciones+1];
```

```
%Eliminar ciudad de su posición
```

```
Ruta=Hijo;
```

```
Premove=find(Hijo==i);
```

```
Ruta(Premove)=[];
```

```
%Ajustar las posiciones de inserción según la eliminación
```

```
posiciones(posiciones > Premove) = posiciones(posiciones > Premove) - 1;
```

```
% Insertar el elemento en las nuevas posiciones (concatenación)
```

```
Ruta1 = [Ruta(1:posiciones(1)-1), i, Ruta(posiciones(1):end)];
```

```
Ruta2 = [Ruta(1:posiciones(2)-1), i, Ruta(posiciones(2):end)];
```

```
%seleccionar la mejor ruta entre: Hijo, Ruta1, Ruta2
```

```
[FOhijo]=FO_TSP(Hijo,d,numCiudades);
```

```
[FORuta1]=FO_TSP(Ruta1,d,numCiudades);
```

```
[FORuta2]=FO_TSP(Ruta2,d,numCiudades);
```

```
%sustitución
```

```
individuos=[Hijo;Ruta1;Ruta2];
```

```
aptitudes=[FOhijo;FORuta1;FORuta2];
```

```
[~,idx]=sort(aptitudes);
```

```
Hijo=individuos(idx(1),:);
```

```
end
```

```
end
```

```
function [vfo]=FO_TSP(ruta,d,numCiudades)
vfo=0;
for i=1:numCiudades
%indicadores de posición
idx1=i;
%el último elemento conecta con el primero
idx2 = (i+1) * (i+1 <= numCiudades) + 1 * (i+1 > numCiudades);
%FO
vfo=vfo+d(ruta(idx1),ruta(idx2));
end
```

# PSEUDOCÓDIGO DEL ALGORITMO GENÉTICO HÍBRIDO (HGA)

## Paso 1:

- Generación aleatoria de la población inicial.
- Evaluar la aptitud de la población inicial

## Paso 2:

- Aplicar el algoritmo "Remoción de Abruptos" a todas las rutas en la población inicial.

## Paso 3:

- Selección de padres aleatoriamente.
- Aplicar el cruce por " Edge Recombination (ER)" entre los padres y generar **un** descendiente.
- Evaluar la aptitud del descendiente.
- Aplicar la heurística "Remoción de Abruptos" al descendiente.

## Paso 4

- Ordenar los padres y el descendiente de acuerdo a su aptitud (Familia ), y pasan a conformar la siguiente generación los dos mejores individuos.



## Paso 5:

- Mezclar aleatoriamente una ruta seleccionada al azar de la población.

Si  $\text{rand} < \text{pm}$  (para cada individuo)  
Se genera un individuo aleatoriamente y se  
sustituye en un miembro de la población aleatorio  
end

## Paso 6:

- Repetir los pasos 3, 4, y 5 hasta el final del número especificado de generaciones.

Ciclo  
principal





# TSP: RUTA DE LAS METRÓPOLIS

Un desafío para encontrar la ruta más eficiente entre las principales ciudades de Estados Unidos, teniendo en cuenta las diversas características y distancias entre ellas.

Ciudad	New York	Los Angeles	Chicago	Houston	Phoenix	Philadelphia	San Diego	Dallas	San Francisco	Austin	Las Vegas
1. New York	-	3091	927	1876	2704	94	2999	1641	3471	1838	3013
2. Los Angeles	3091	-	2542	1681	375	2994	138	1442	389	1407	290
3. Chicago	927	2542	-	1337	2169	930	2464	1100	2935	1168	2465
4. Houston	1876	1681	1337	-	1308	1778	1603	240	2075	163	1604
5. Phoenix	2704	375	2169	1308	-	2603	366	1069	767	1034	296
6. Philadelphia	94	2994	930	1778	2603	-	2898	1543	3369	1740	2915
7. San Diego	2999	138	2464	1603	366	2898	-	1364	531	1329	338
8. Dallas	1641	1442	1100	240	1069	1543	1364	-	1836	201	1365
9. San Francisco	3471	389	2935	2075	767	3369	531	1836	-	1801	607
10. Austin	1838	1407	1168	163	1034	1740	1329	201	1801	-	1330
11. Las Vegas	3013	290	2465	1604	296	2915	338	1365	607	1330	-