

Backtracking

Branch & bound

Búsqueda exhaustiva



krosseel

Algoritmo sencillo para un
problema de búsqueda:

Probar cada una de las posibles soluciones

En el caso típico, el espacio de búsqueda es demasiado grande.

Búsqueda exhaustiva



krosseel

Algoritmo sencillo para un problema de búsqueda:

Probar cada una de las posibles soluciones

En el caso típico, el espacio de búsqueda es demasiado grande.

- ▶ Si hemos olvidado la combinación de este candado (y no podemos recurrir a ningún atajo), ¿con cuántas tendremos que probar hasta abrirlo, en el caso peor?

Búsqueda exhaustiva



Algoritmo sencillo para un
problema de búsqueda:

Probar cada una de las posibles soluciones

En el caso típico, el espacio de búsqueda es demasiado grande.

- ▶ Si hemos olvidado la combinación de este candado (y no podemos recurrir a ningún atajo), ¿con cuántas tendremos que probar hasta abrirlo, en el caso peor?
 $10^4 = 10\,000$: desde 0000 hasta 9999.

Comprobación de las soluciones parciales



krosseel

La tarea de abrir este candado podría ser muy sencilla.

Comprobación de las soluciones parciales



La tarea de abrir este candado podría ser muy sencilla.

Si, conocidos los primeros dígitos de la combinación, existe alguna manera de determinar si el siguiente es o no el correcto, la cantidad de intentos necesarios se reduce drásticamente.

- ▶ En el caso peor, $10 + 10 + 10 + 10 = 40$.

Backtracking

Este es el fundamento del *backtracking*: la posibilidad de abandonar (cuanto antes, mejor) algunas ramas del árbol de opciones de una búsqueda exhaustiva.

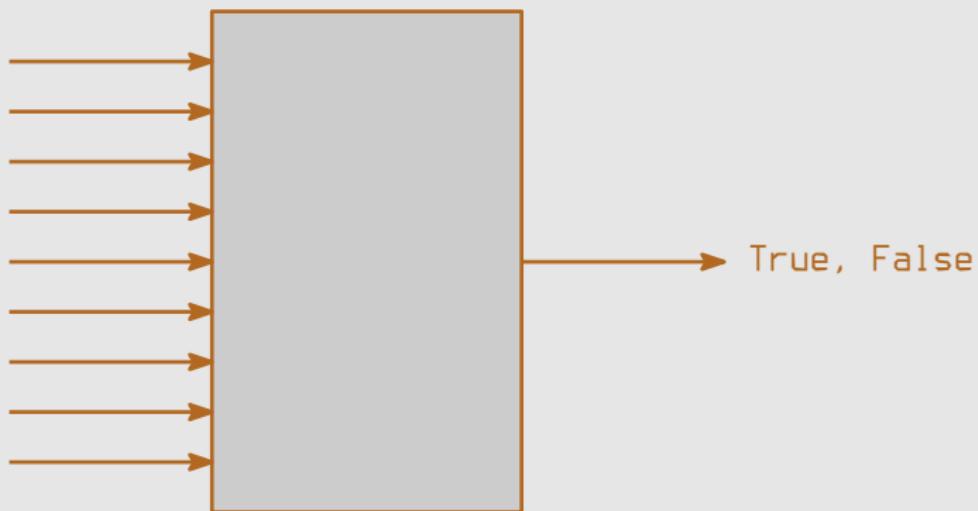
El término hace referencia a la «vuelta atrás» al encontrar un camino sin salida viable.

D. H. Lehmer, en los años cincuenta, comenzó a utilizar esta palabra para denominar esta técnica.

Backtracking

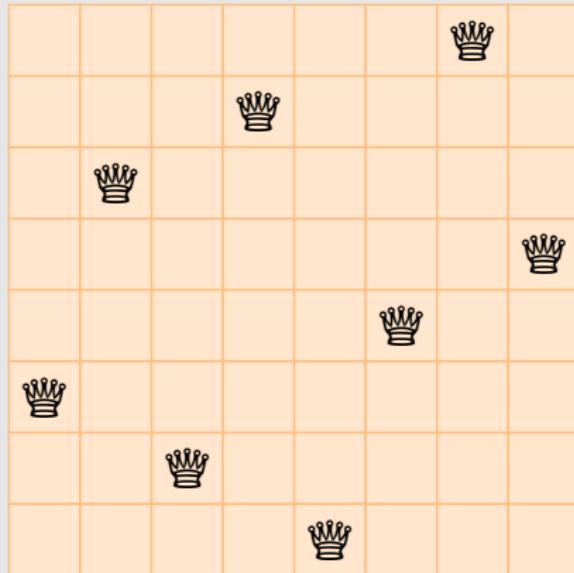
En ocasiones, no se puede evaluar una función con múltiples entradas sin conocerlas todas.

La técnica del *backtracking* se puede aplicar si, en algunos casos, es suficiente contar con información parcial para garantizar un resultado negativo.



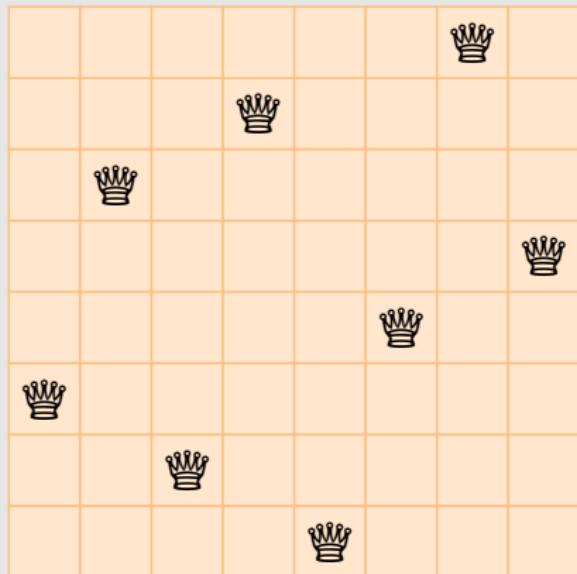
Reinas en tregua

¿Cómo colocar ocho reinas en un tablero de ajedrez sin que ninguna amenace a otra?



Reinas en tregua

¿Cómo colocar ocho reinas en un tablero de ajedrez sin que ninguna amenace a otra?



En general, ¿cómo colocar n reinas en un tablero de tamaño $n \times n$?

Reinas en tregua

- ▶ La estimación más grosera para el espacio de búsqueda es

$$\binom{n^2}{n} \xrightarrow{n=8} 4\,426\,165\,368.$$

Reinas en tregua

- ▶ La estimación más grosera para el espacio de búsqueda es

$$\binom{n^2}{n} \xrightarrow{n=8} 4\,426\,165\,368.$$

- ▶ Teniendo en cuenta que no puede haber dos reinas en una misma fila, el espacio se reduce a

$$n^n \xrightarrow{n=8} 16\,777\,216.$$

Reinas en tregua

- ▶ La estimación más grosera para el espacio de búsqueda es

$$\binom{n^2}{n} \xrightarrow{n=8} 4426\,165\,368.$$

- ▶ Teniendo en cuenta que no puede haber dos reinas en una misma fila, el espacio se reduce a

$$n^n \xrightarrow{n=8} 16\,777\,216.$$

- ▶ Considerando, además, que tampoco puede repetirse una columna, reducimos el ámbito de búsqueda a las *matrices permutación*:

$$n! \xrightarrow{n=8} 40\,320.$$

Reinas en tregua

- ▶ La estimación más grosera para el espacio de búsqueda es

$$\binom{n^2}{n} \xrightarrow{n=8} 4426165368.$$

- ▶ Teniendo en cuenta que no puede haber dos reinas en una misma fila, el espacio se reduce a

$$n^n \xrightarrow{n=8} 16777216.$$

- ▶ Considerando, además, que tampoco puede repetirse una columna, reducimos el ámbito de búsqueda a las *matrices permutación*:

$$n^n e^{-n} \sqrt{2\pi n} \sim n! \xrightarrow{n=8} 40320.$$

Fórmula de Stirling

Reinas en tregua

Un algoritmo elemental de **búsqueda exhaustiva** permite resolver el problema para tamaños pequeños (incluyendo $n = 8$).

— dia_00_cod_01.py ————— dia_00_cod_02.py —————

```
# Problema de búsqueda          # Problema de enumeración

def tregua(n):
    for p in opciones(n):
        if not p.conflicto():
            return p

def tregua(n):
    # sols = list()
    # for p in opciones(n):
    #     if not p.conflicto():
    #         sols.append(p)

    sols = [p for p in opciones(n) \
            if not p.conflicto()]

    return sols
```

► Con backtracking

Reinas en tregua

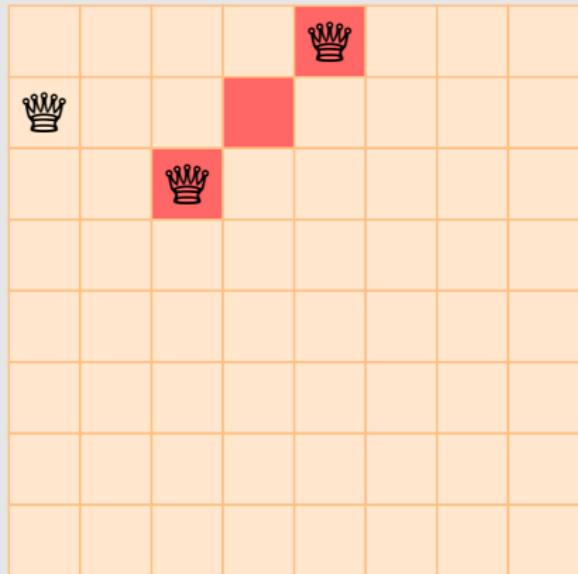
n	1	2	3	4	5	6	7	8
	1	0	0	2	10	4	40	92

<http://oeis.org/A000170>



Reinas en tregua

Este es un problema típico en el que se pueden **descartar soluciones parciales.**

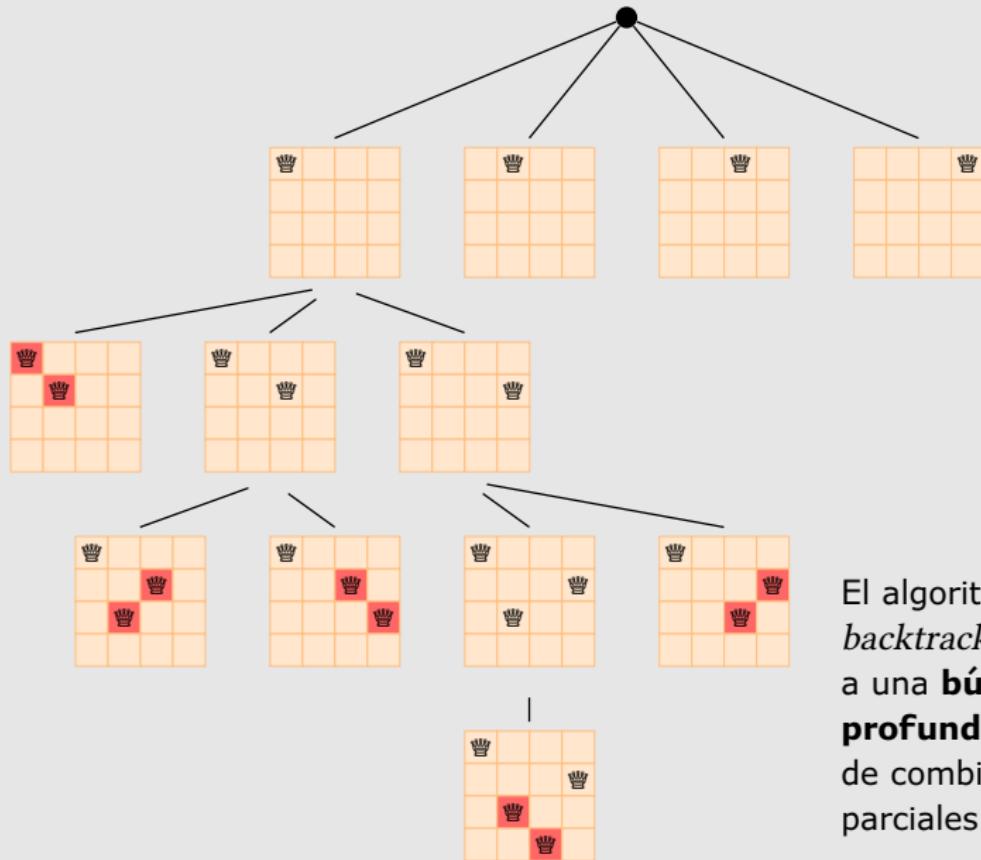


No es necesario construir todas las posibles configuraciones.

Al completar el *árbol* de posibilidades, algunas ramas se pueden abandonar.

Esta técnica se conoce como **backtracking**.

Reinas en tregua



El algoritmo de *backtracking* se asemeja a una **búsqueda en profundidad** en un árbol de combinaciones parciales.

Reinas en tregua

_____ dia_00_cod_05.py _____

```
# Problema de búsqueda

def tregua(n):
    p = Parcial(n)
    return desciende(p)

def desciende(p):
    if p.completa():
        return p
    else:
        for pa in p.amplía():
            if not pa.conflicto():
                aux = desciende(pa)
                if aux:
                    return aux
```

_____ dia_00_cod_04.py _____

```
# Problema de enumeración

def tregua(n):
    sols = list()
    p = Parcial(n)
    desciende(p, sols)
    return sols

def desciende(p, sols):
    if p.completa():
        sols.append(p)
    else:
        for pa in p.amplía():
            if not pa.conflicto():
                desciende(pa, sols)
```

Reinas en tregua

Compara el rendimiento de estos algoritmos con la búsqueda exhaustiva «elemental» que planteábamos antes.

► Búsqueda exhaustiva

Problemas de búsqueda y enumeración

Cada opción del conjunto a explorar puede ser o no una solución válida:

$$f : U \longrightarrow \{\text{True}, \text{False}\}$$

- ▶ **Búsqueda:** encontrar $x \in f^{-1}(\text{True})$
o determinar que no lo hay ($\emptyset = f^{-1}(\text{True})$)

- ▶ **Enumeración:** calcular $|f^{-1}(\text{True})|$
(o, más aún, describir sus elementos)

El conjunto U es finito.

Problema de optimización

Cada opción del conjunto a explorar se evalúa con cierta medida:

$$f : U \longrightarrow \mathbb{N} \quad (\text{o } \mathbb{Z}, \mathbb{R} \dots)$$

- ▶ **Optimización:** encontrar $x \in U$ tal que $f(x) = \min(f)$

El conjunto U es finito.

Problema de optimización

Cada opción del conjunto a explorar se evalúa con cierta medida:

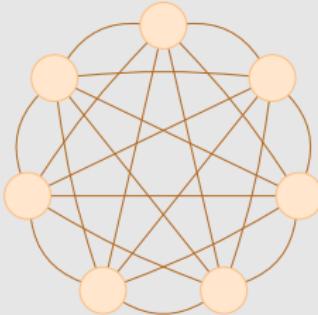
$$f : U \longrightarrow \mathbb{N} \quad (\text{o } \mathbb{Z}, \mathbb{R} \dots)$$

- ▶ **Optimización:** encontrar $x \in U$ tal que $f(x) = \min(f)$

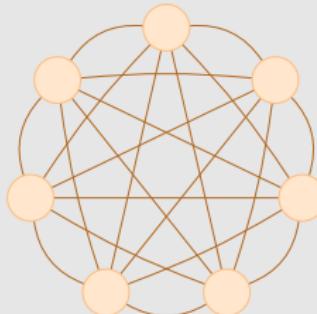
Conceptualmente, es indiferente referirse al mínimo o al máximo.

El conjunto U es finito.

- ▶ Se refiere a un **grafo completo** no dirigido (K_n).



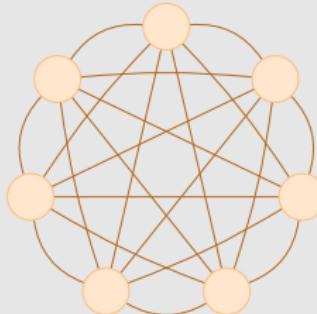
- ▶ Se refiere a un **grafo completo** no dirigido (K_n).



- ▶ Busca un circuito *hamiltoniano*.

camino cerrado que visita todos los vértices sin repetir ninguno

- ▶ Se refiere a un **grafo completo** no dirigido (K_n).

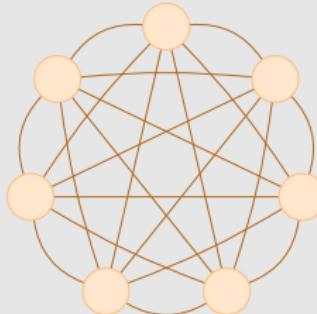


- ▶ Busca un circuito *hamiltoniano*.

camino cerrado que visita todos los vértices sin repetir ninguno

¿Hasta aquí es fácil?

- ▶ Se refiere a un **grafo completo** no dirigido (K_n).



- ▶ Busca un circuito *hamiltoniano*.

camino cerrado que visita todos los vértices sin repetir ninguno

¿Hasta aquí es fácil?

Hay $(n-1)!/2$ circuitos hamiltonianos en K_n (si $n \geq 3$).

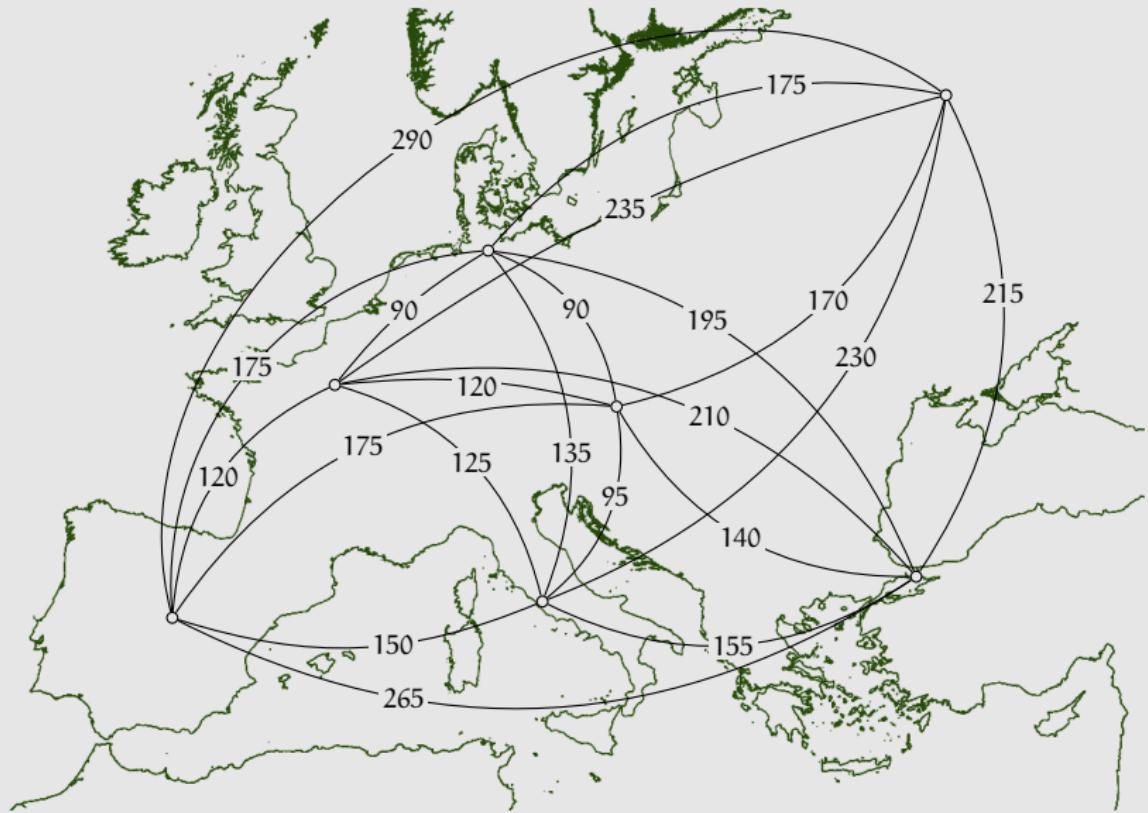
Pero este problema no se contenta con encontrar un circuito hamiltoniano cualquiera.



Problema del viajante

Travelling salesman problem

Las aristas del grafo están gravadas con ciertos **pesos**.



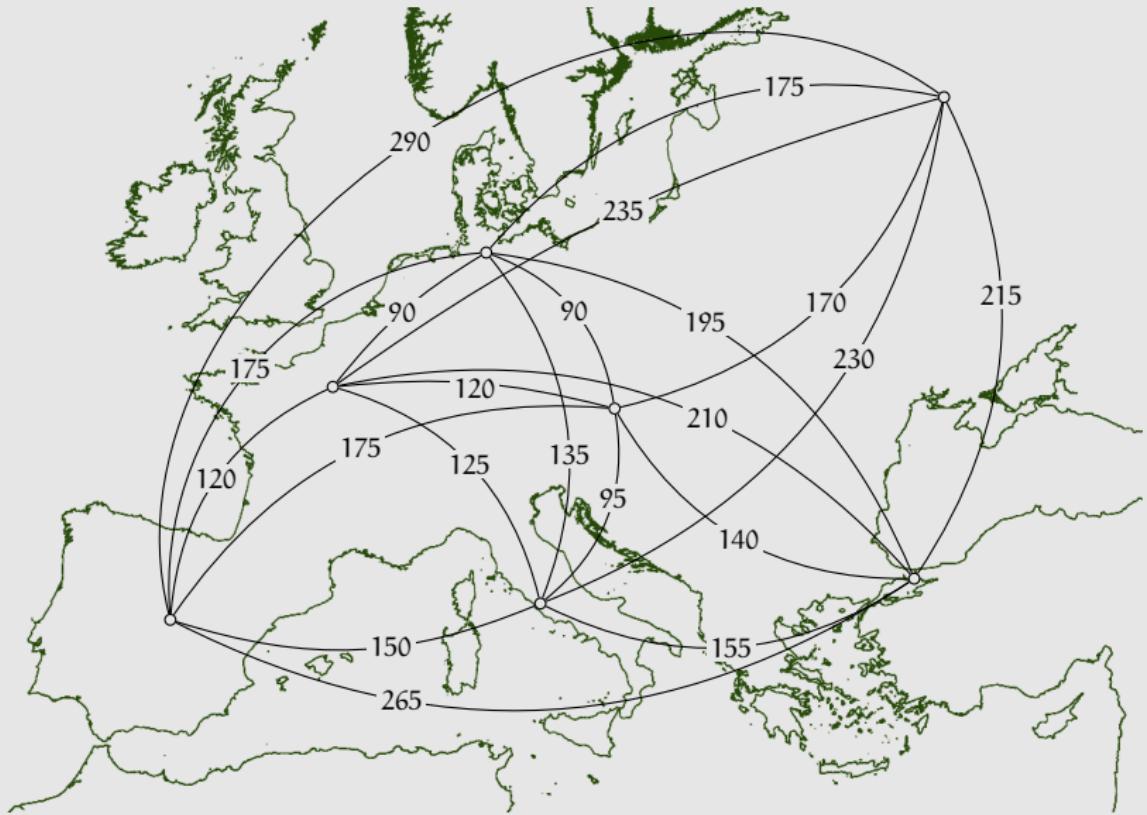
En este ejemplo, tiempos de vuelo.



Problema del viajante

Travelling salesman problem

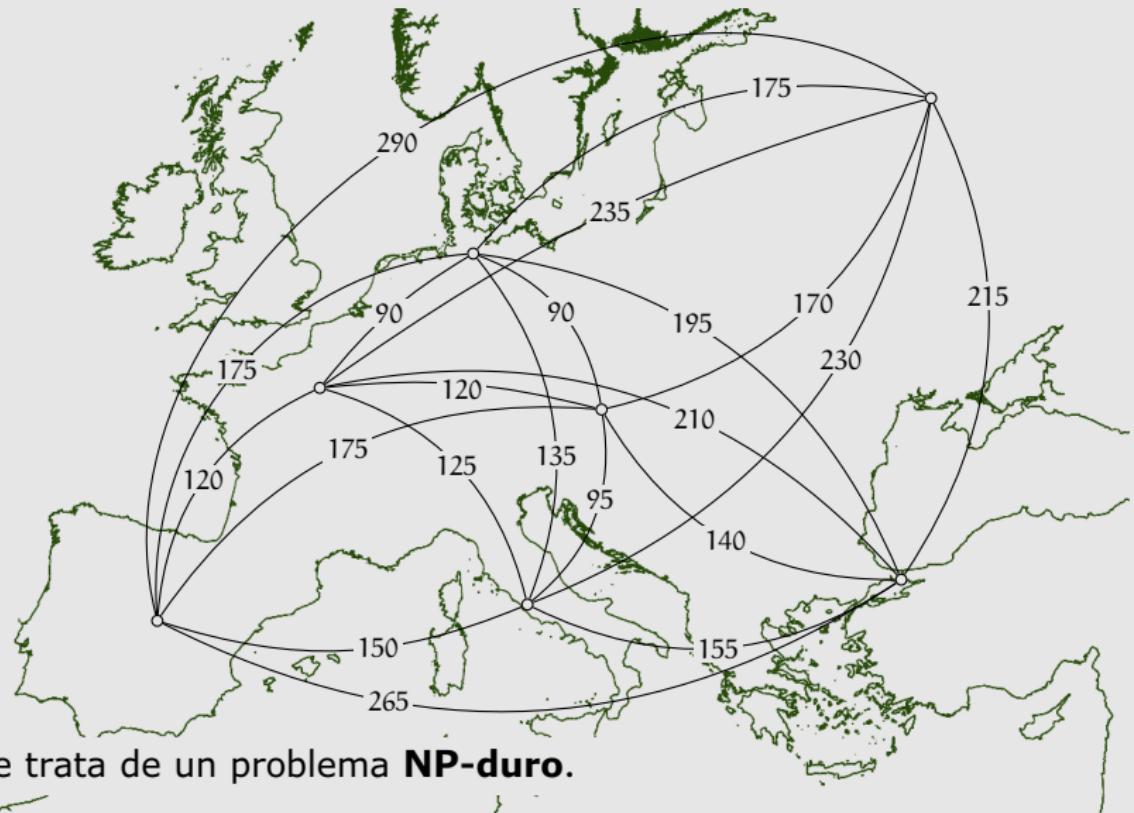
El objetivo es **minimizar** el coste del circuito.



Problema del viajante

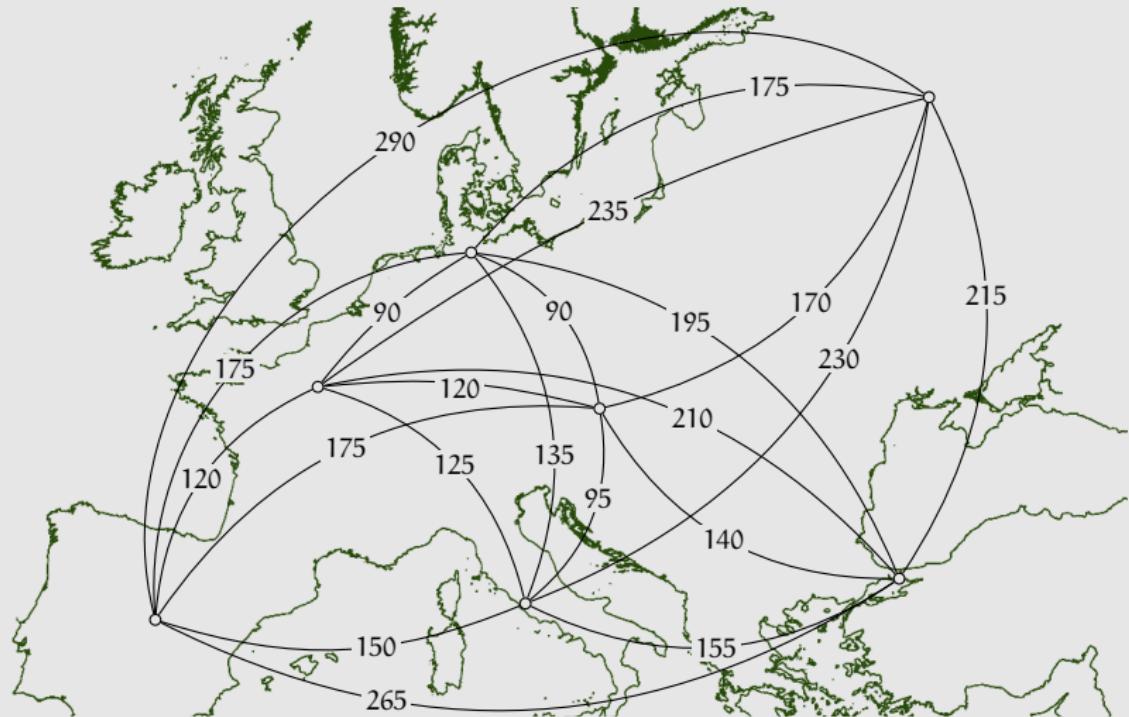
Travelling salesman problem

El objetivo es **minimizar** el coste del circuito.



Se trata de un problema **NP-duro**.

El objetivo es **minimizar** el coste del circuito.



En su versión decisional (¿existe un circuito que no tarde más de T?), es NP,
y **NP-completo**.

dia_00_cod_21.py

```
def ordenados():
    lista = [(cic, cic.coste())\
              for cic in dt.ciclos('datos.txt')]

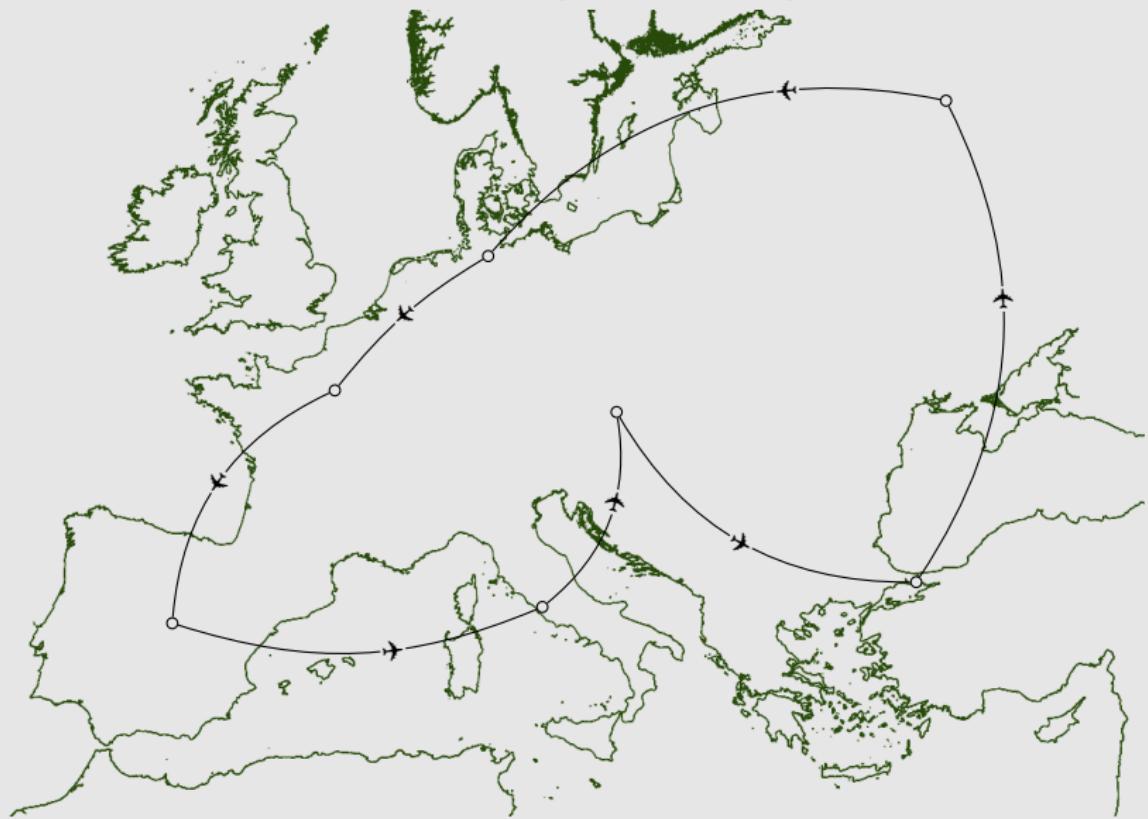
    # Ordena la lista.

    lista.sort(key=lambda tupla: tupla[1])

    # Presenta los resultados.

    m = len(str(lista[-1][1]))
    f = open('ordenados.txt', 'w')
    for cic, t in lista:
        f.write(str(cic) + ':\\t{0:>{1}d}\\n'.format(t, m))
    f.close()
    cic, t = lista[0]
    print(cic, '\\t', t, 'minutos')
```

Este es el circuito menos costoso (985 minutos).



Seguido de cerca por este otro (990 minutos).



Este recorrido no parece muy cabal.



Problema del viajante:

¿backtracking?

Con problemas como este, no podemos desechar sin más una solución parcial.

Problema del viajante:

¿backtracking?

Con problemas como este, no podemos desechar sin más una solución parcial.

Existe, sin embargo, una técnica similar conocida como **branch and bound** (ramificación y poda).

Con problemas como este, no podemos desechar sin más una solución parcial.

Existe, sin embargo, una técnica similar conocida como **branch and bound** (ramificación y poda).

Se basa en una **acotación inferior** de la medida de las soluciones derivadas de una parcial.

Si la cota es mayor o igual que el coste de la mejor solución hallada hasta el momento, la solución parcial se puede descartar, porque de ella no derivan soluciones mejores.

Problema del viajante:

branch & bound

Supongamos fijada cierta parte del recorrido, p. ej., MAD-FCO-VIE.



Problema del viajante:

branch & bound

Supongamos fijada cierta parte del recorrido, p. ej., MAD-FCO-VIE.

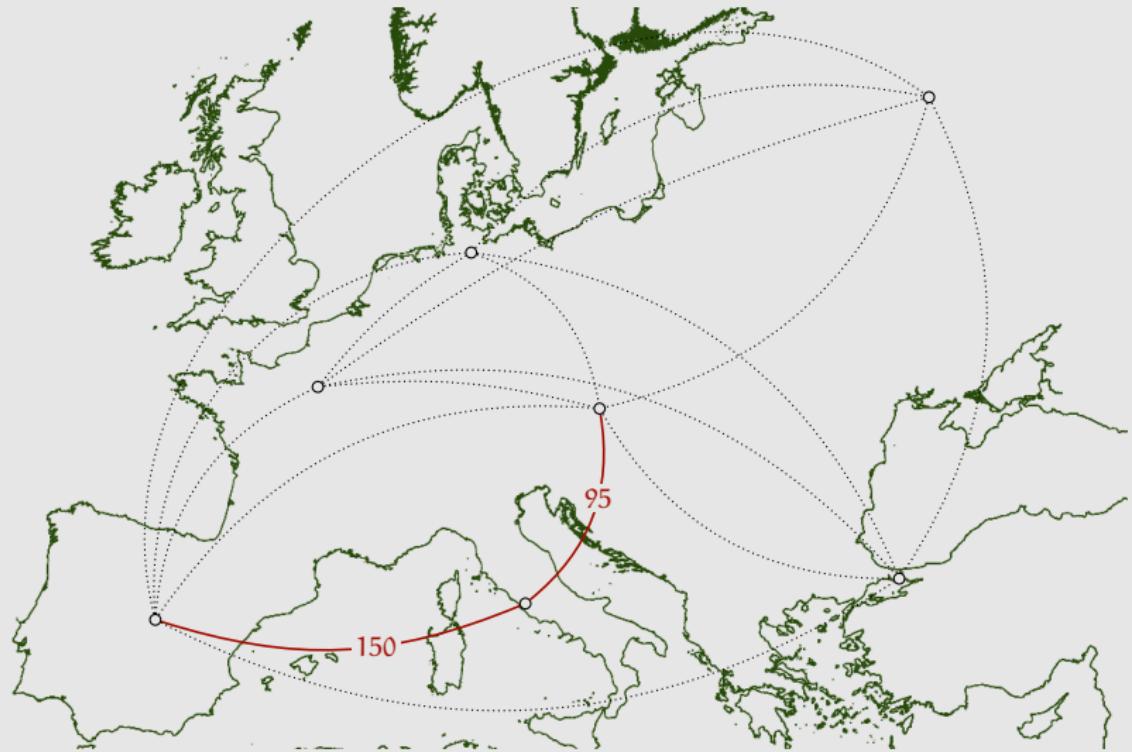


Todo ciclo ham. que contenga estos tramos cuesta, **como poco...** ¿cuánto?

Problema del viajante:

branch & bound

Supongamos fijada cierta parte del recorrido, p. ej., MAD-FCO-VIE.

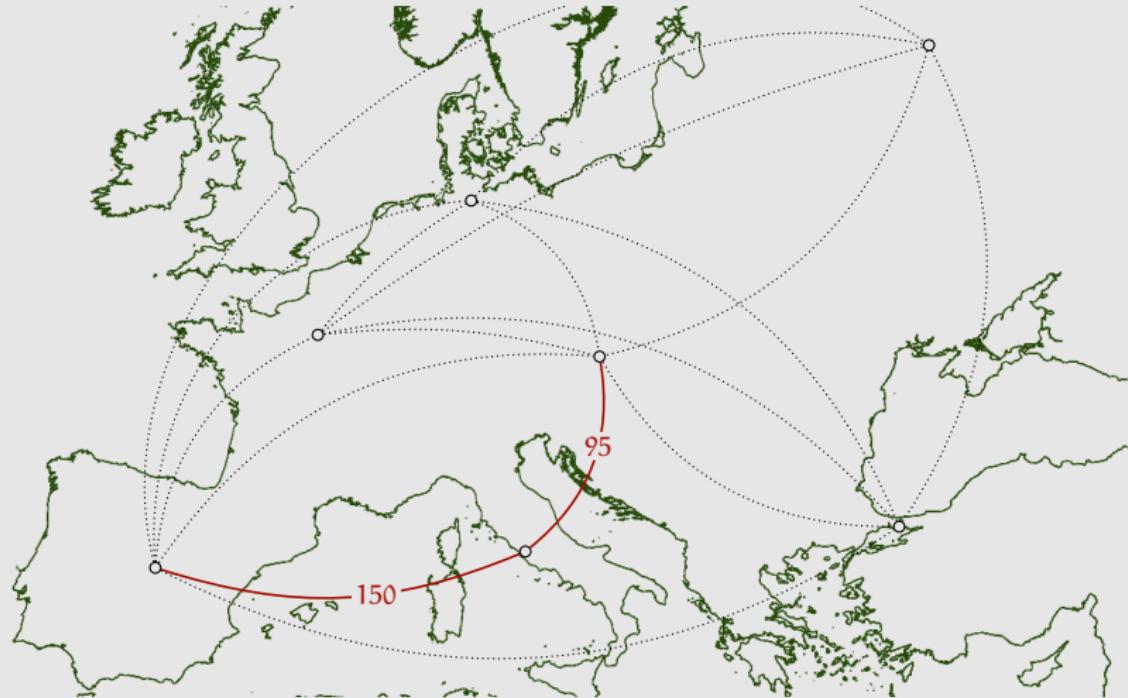


La cota obvia es el coste del camino parcial: 245 minutos.

Problema del viajante:

branch & bound

Pero esta cota es excesivamente holgada y no reduce demasiado el campo de búsqueda.



La cota obvia es el coste del camino parcial: **245 minutos.**

dia_00_cod_23.py

```
class Parcial_ct(Parcial):

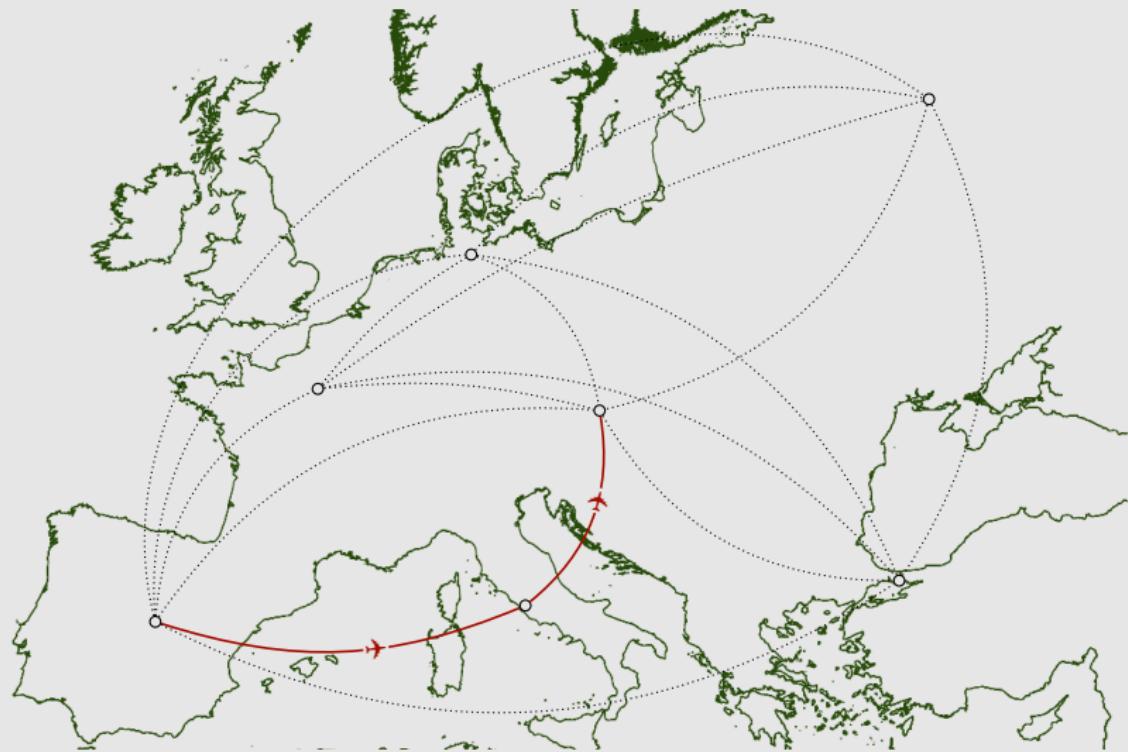
    def cota(self):
        return self.coste()

    def lanza(p, mj=None, t_mj=infinito):
        ct = p.cota()
        fondo = p.es_completa()
        escribe_línea(p, ct, t_mj, fondo, mejora)
        if ct < t_mj:
            if fondo:
                mj = p
                t_mj = ct
            else:
                for ap in p.amplía():
                    mj, t_mj = lanza(ap, mj, t_mj)
        return mj, t_mj
```

Problema del viajante:

branch & bound

Supongamos fijada cierta parte del recorrido, p. ej., **MAD-FCO-VIE**.

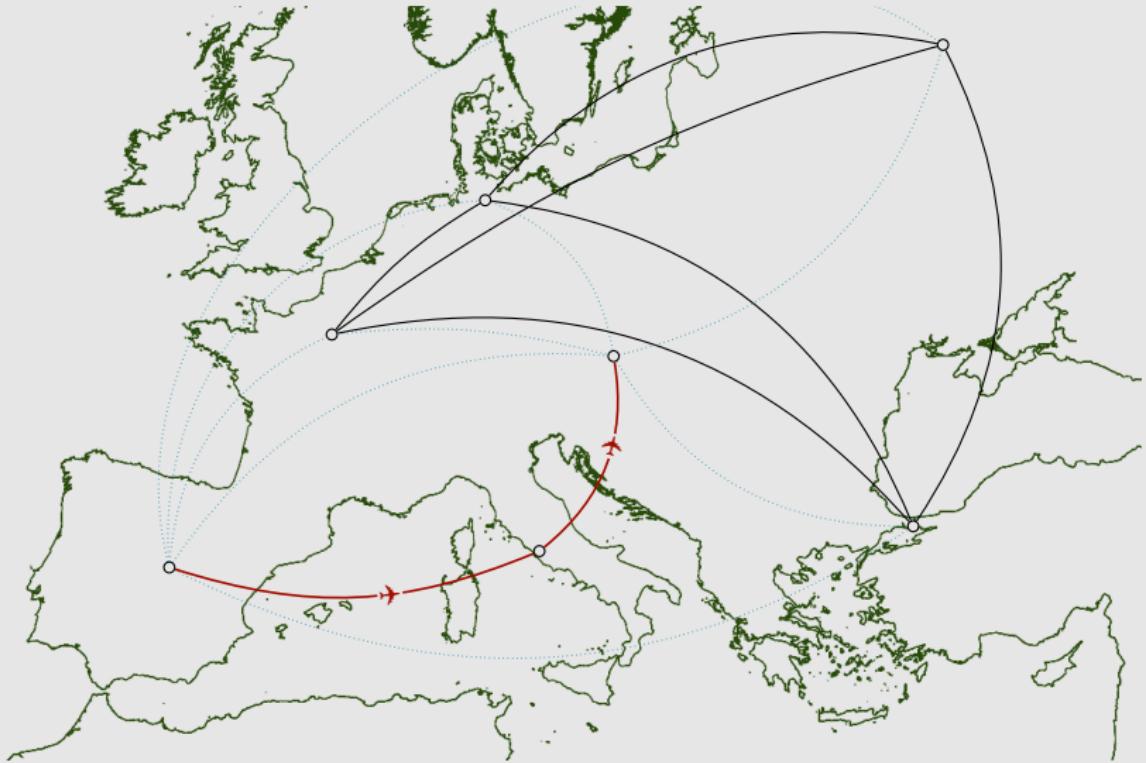


Todo ciclo ham. que contenga estos tramos cuesta, **como poco...** ¿cuánto?

Problema del viajante:

branch & bound

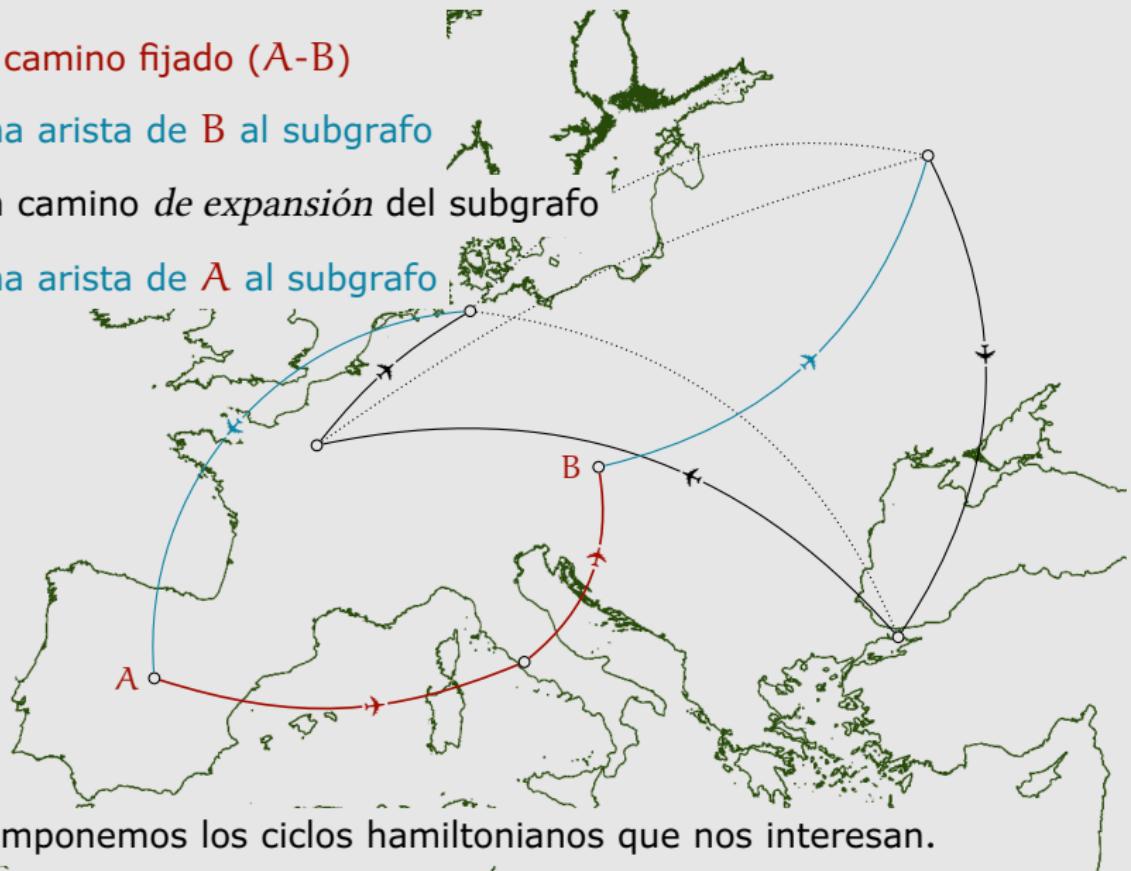
Consideramos el subgrafo (completo) inducido por los nodos que no están en el camino fijado.



Problema del viajante:

branch & bound

- ▶ el camino fijado (A-B)
- ▶ una arista de B al subgrafo
- ▶ un camino *de expansión* del subgrafo
- ▶ una arista de A al subgrafo

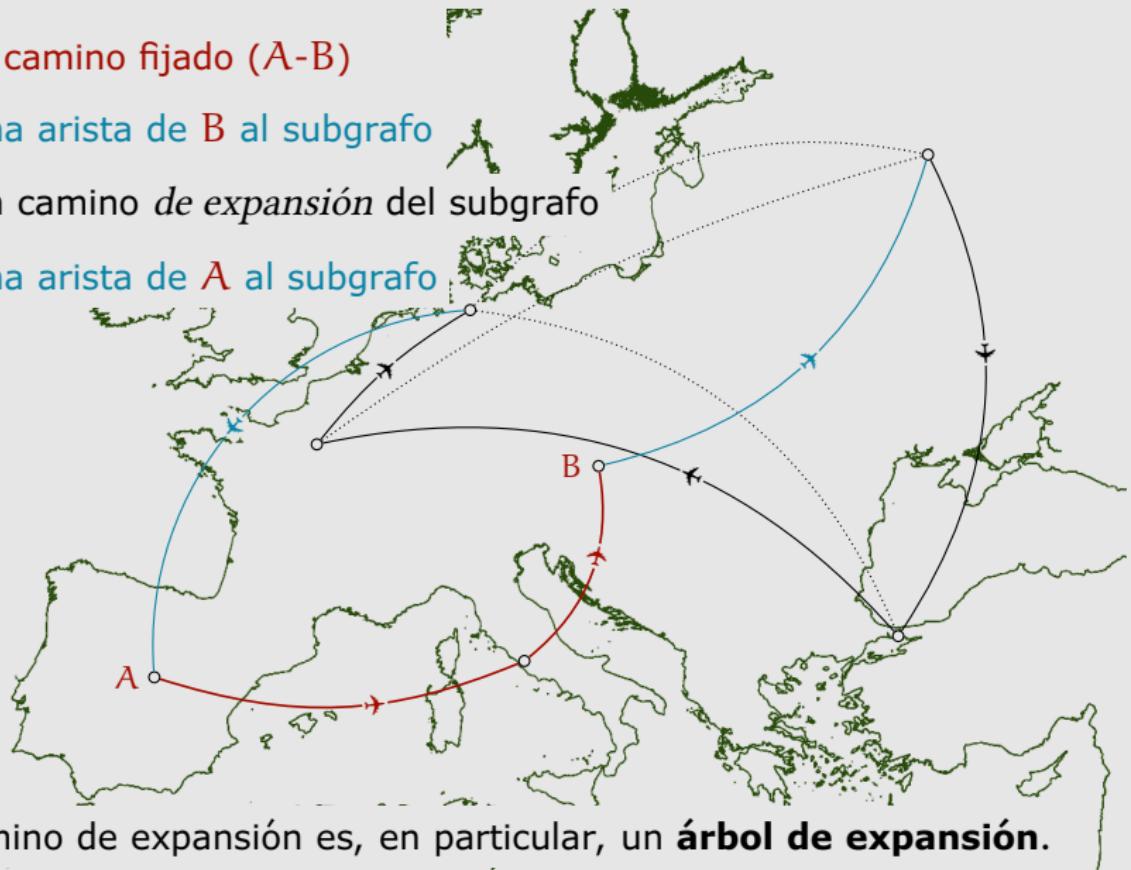


Descomponemos los ciclos hamiltonianos que nos interesan.

Problema del viajante:

branch & bound

- ▶ el camino fijado (A-B)
- ▶ una arista de B al subgrafo
- ▶ un camino *de expansión* del subgrafo
- ▶ una arista de A al subgrafo

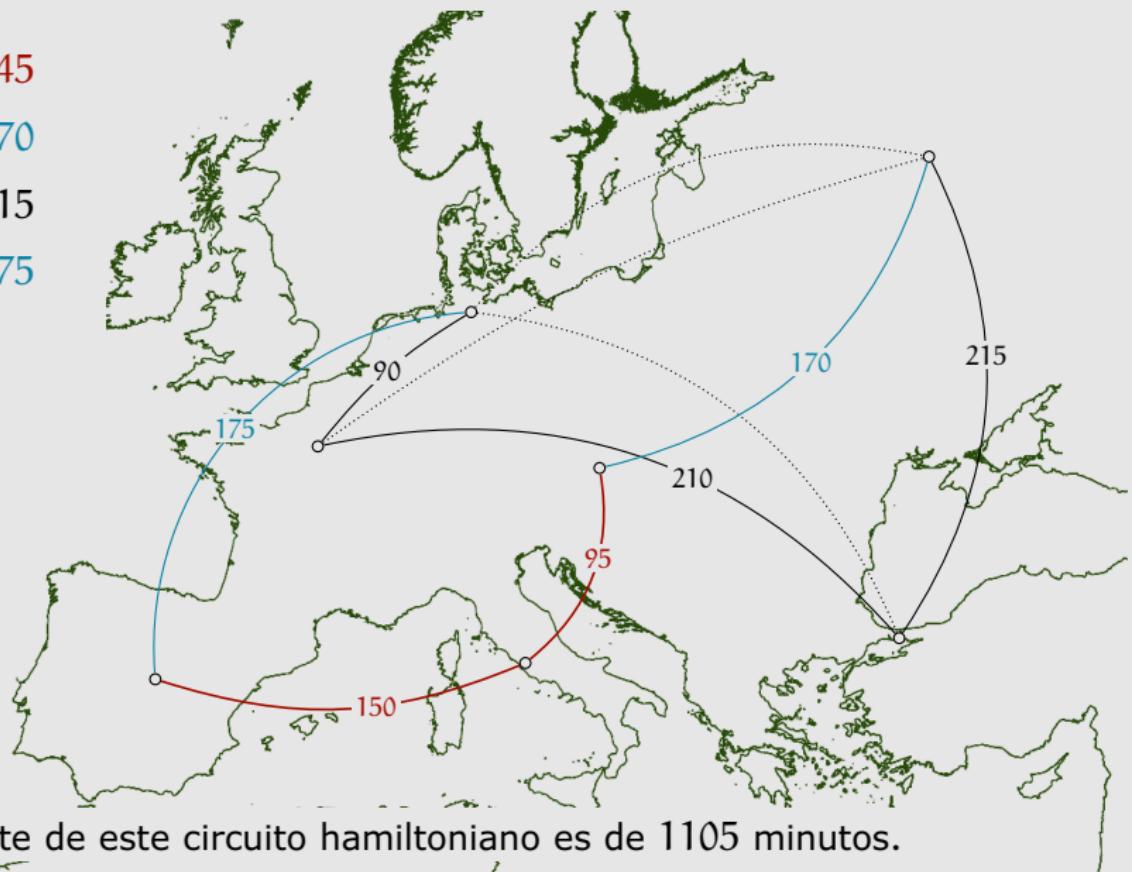


El camino de expansión es, en particular, un **árbol de expansión**.

Problema del viajante:

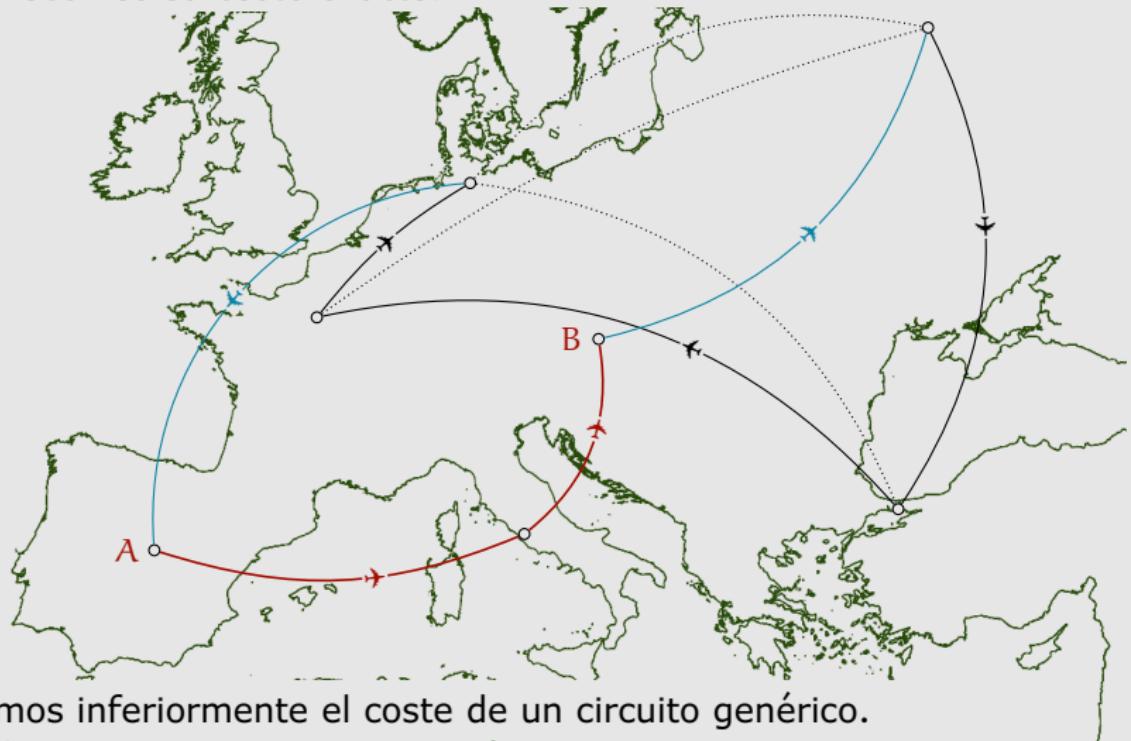
branch & bound

- ▶ 245
- ▶ 170
- ▶ 515
- ▶ 175



► El camino fijado (A-B):

conocemos su coste exacto.

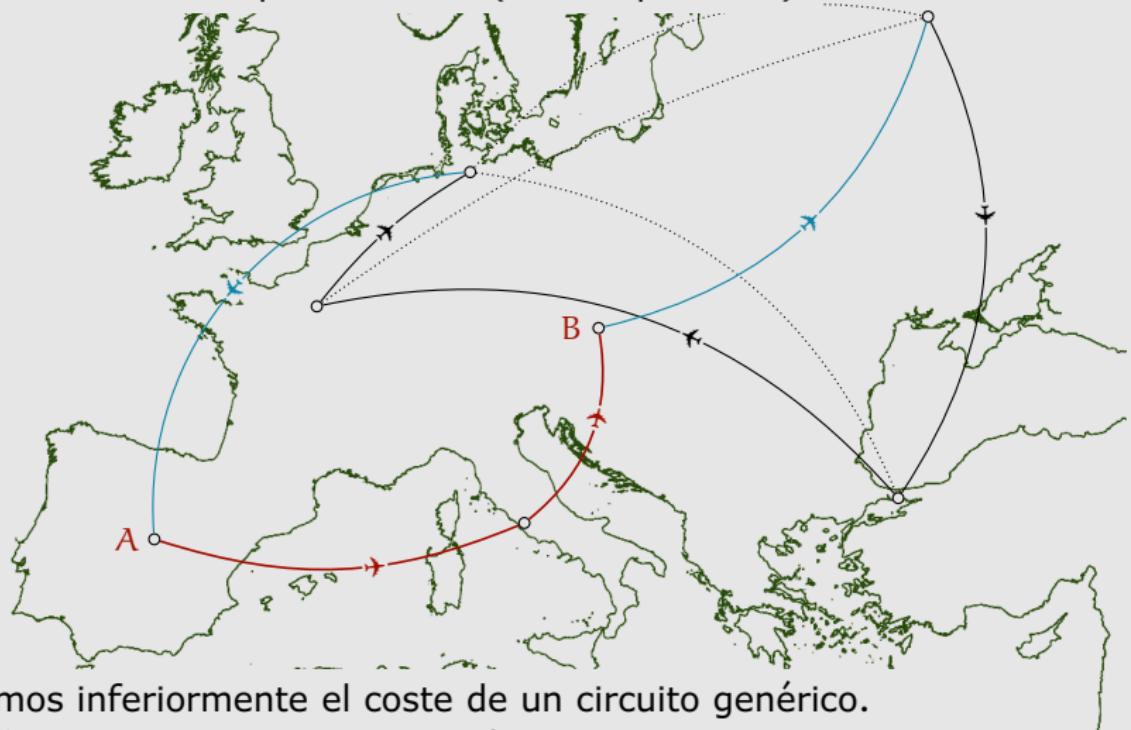


Problema del viajante:

branch & bound

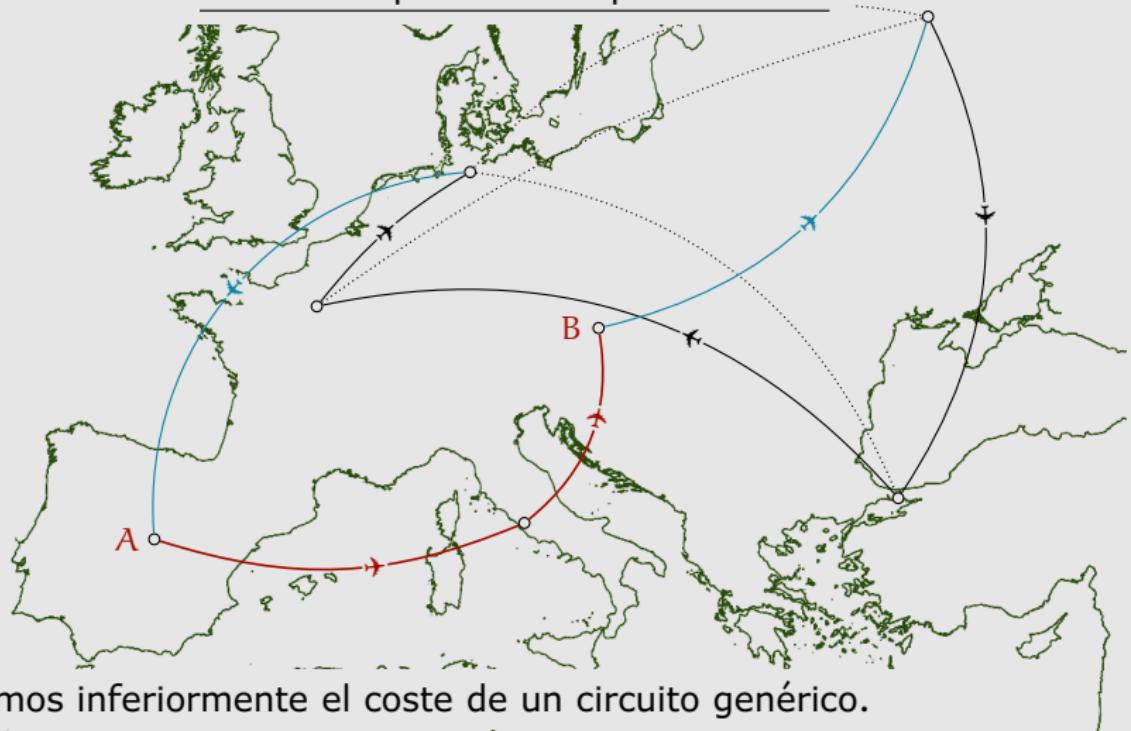
- ▶ Aristas del subgrafo a B y A:

buscamos las de peso mínimo (en tiempo lineal).



Acotemos inferiormente el coste de un circuito genérico.

- ▶ Un camino *de expansión* del subgrafo:
buscamos un árbol de expansión con peso mínimo.



Acotemos inferiormente el coste de un circuito genérico.

Problema del viajante:

branch & bound

Estas elecciones no garantizan un circuito hamiltoniano:

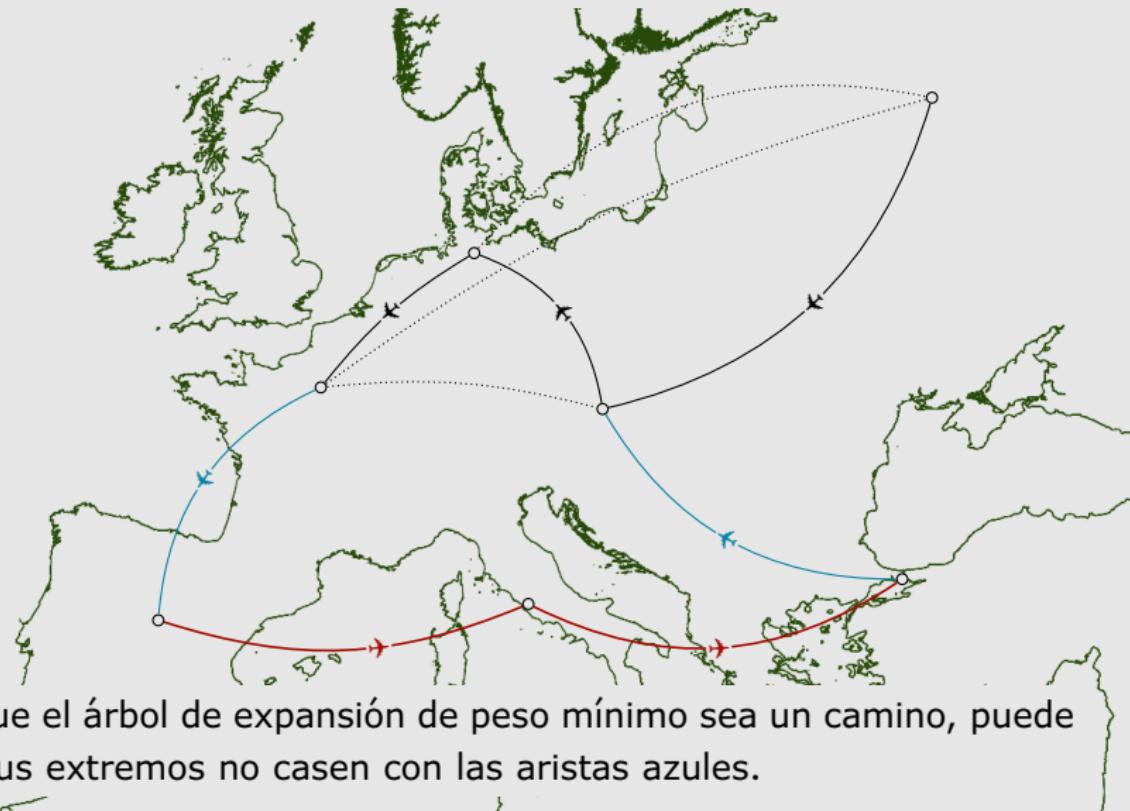


El árbol no tiene por qué ser un camino.

Problema del viajante:

branch & bound

Estas elecciones no garantizan un circuito hamiltoniano:

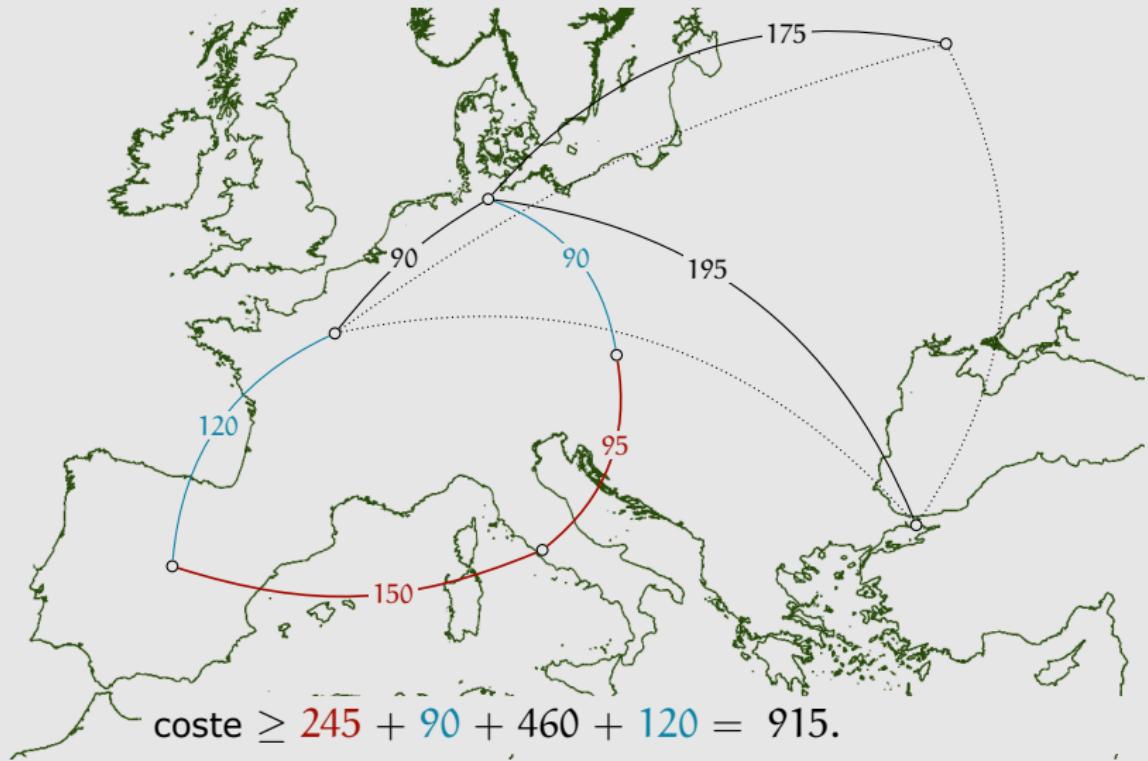


Aunque el árbol de expansión de peso mínimo sea un camino, puede que sus extremos no casen con las aristas azules.

Problema del viajante:

branch & bound

Pero sí garantizan una cota inferior para el coste de los circuitos hamiltonianos que contienen los tramos fijados.



Se recorre el árbol de soluciones parciales mediante una búsqueda en profundidad.

Se recorre el árbol de soluciones parciales mediante una búsqueda en profundidad.

Para cada nodo, se calcula la cota inferior: las soluciones derivadas de ese nodo no son mejores que esa cota.

Se recorre el árbol de soluciones parciales mediante una búsqueda en profundidad.

Para cada nodo, se calcula la cota inferior: las soluciones derivadas de ese nodo no son mejores que esa cota.

Si la cantidad calculada no es menor que la mejor solución completa encontrada, se puede descartar la rama.

Se recorre el árbol de soluciones parciales mediante una búsqueda en profundidad.

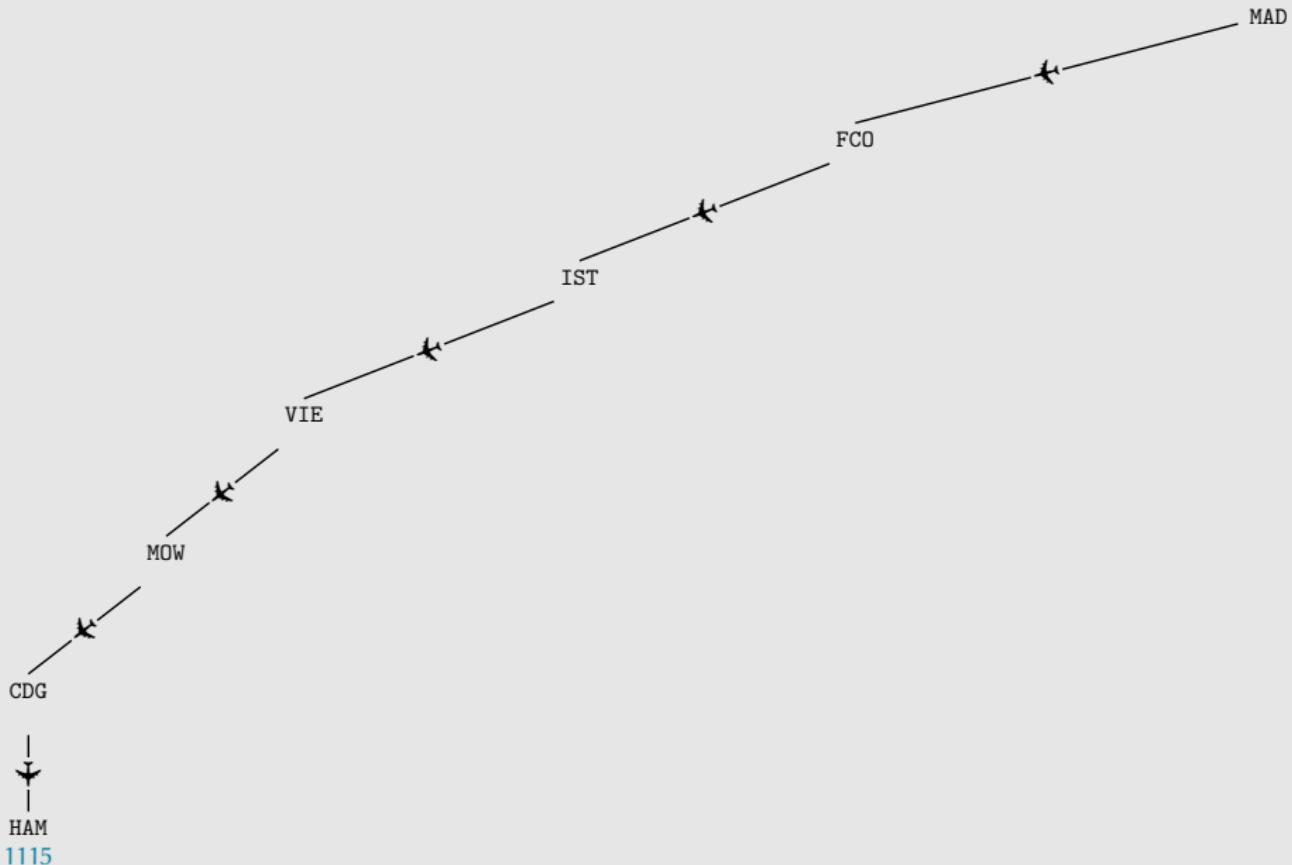
Para cada nodo, se calcula la cota inferior: las soluciones derivadas de ese nodo no son mejores que esa cota.

Si la cantidad calculada no es menor que la mejor solución completa encontrada, se puede descartar la rama.

No es necesario calcular las cotas antes de encontrar la primera solución.

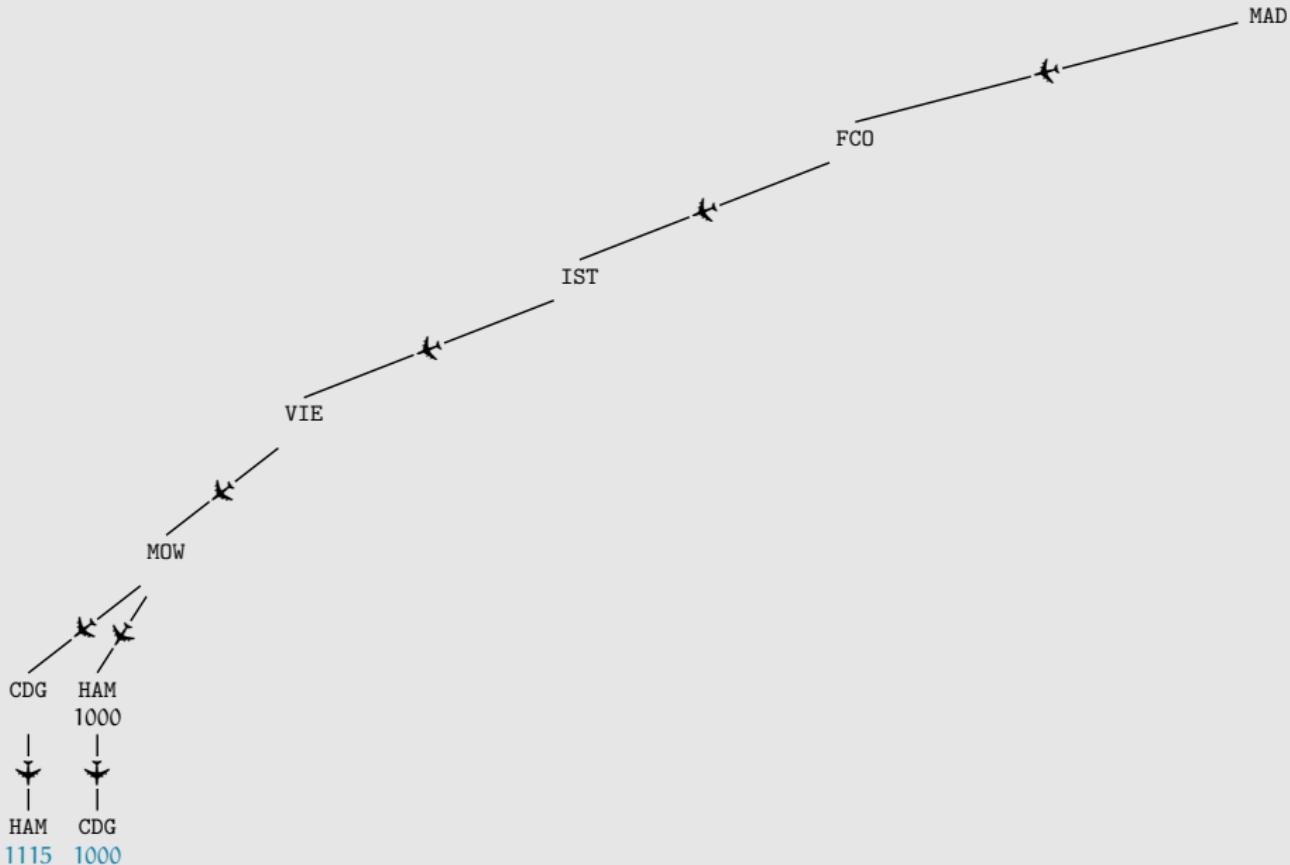
Problema del viajante:

branch & bound



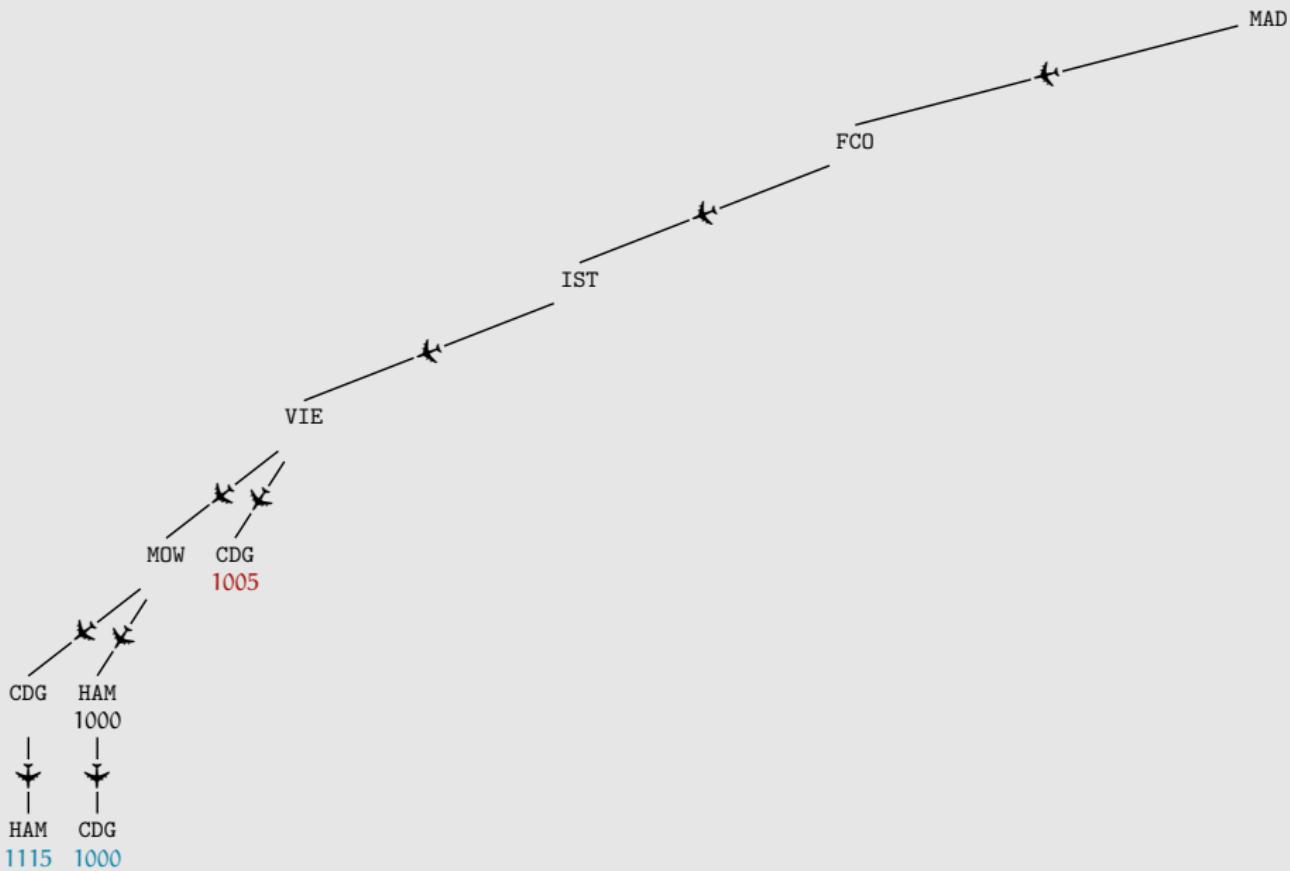
Problema del viajante:

branch & bound



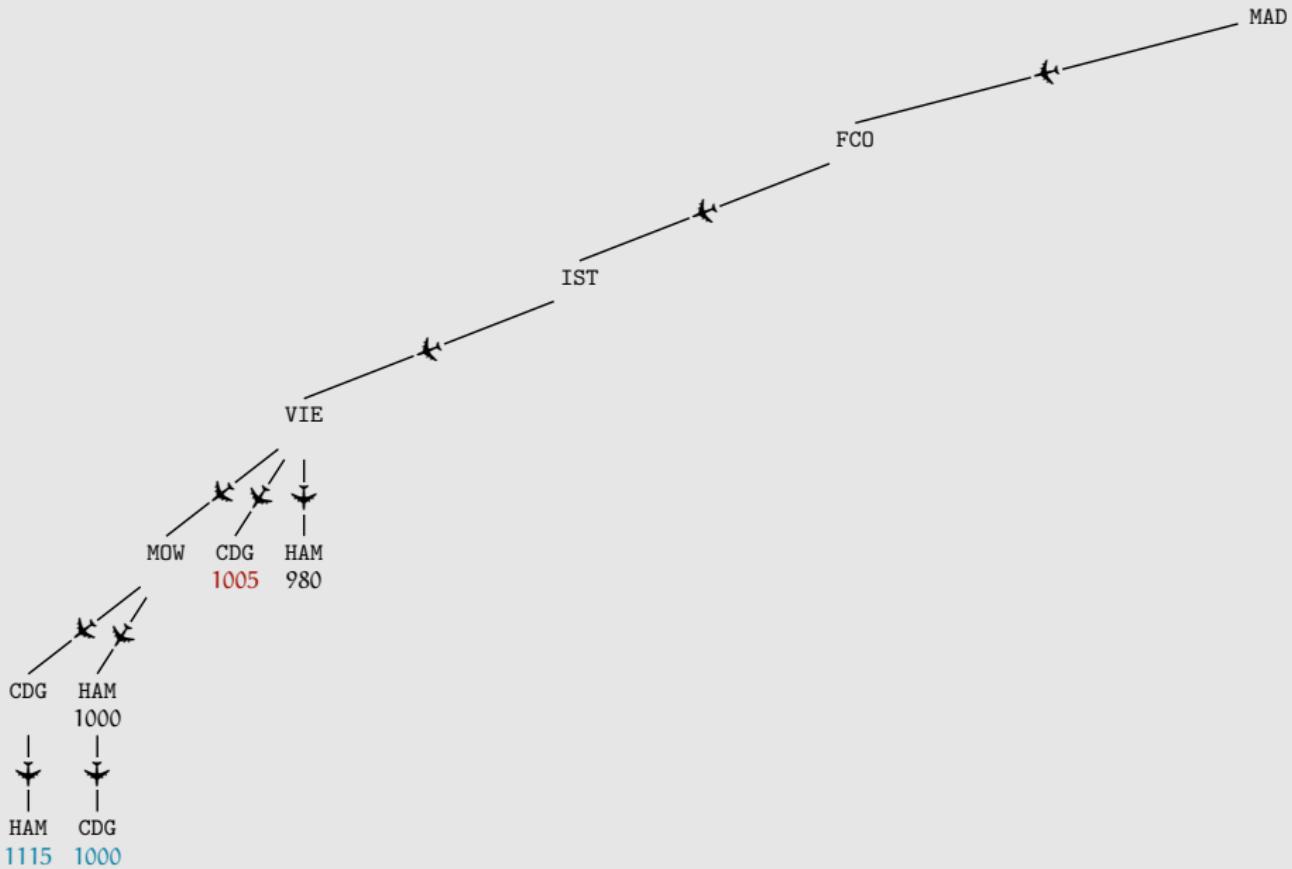
Problema del viajante:

branch & bound



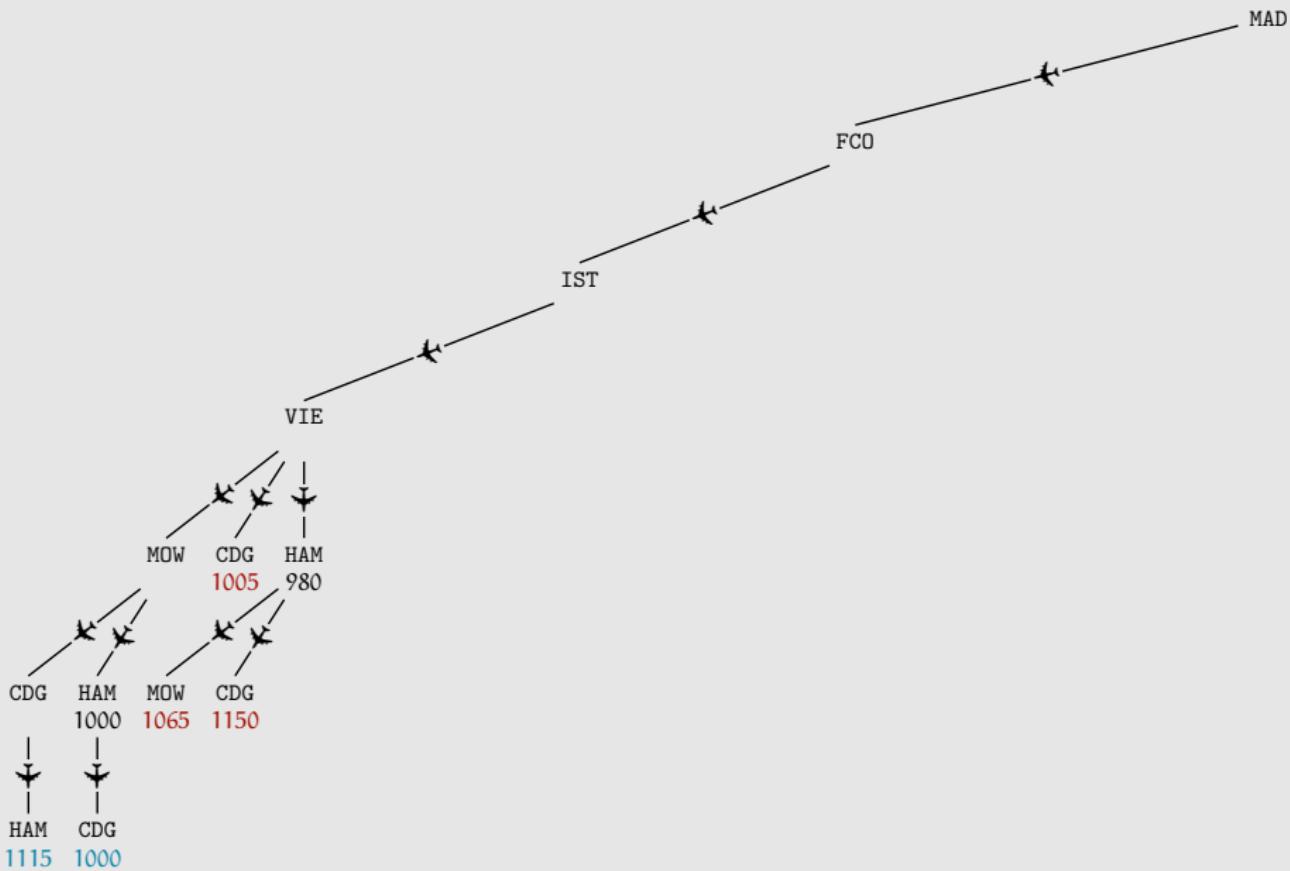
Problema del viajante:

branch & bound



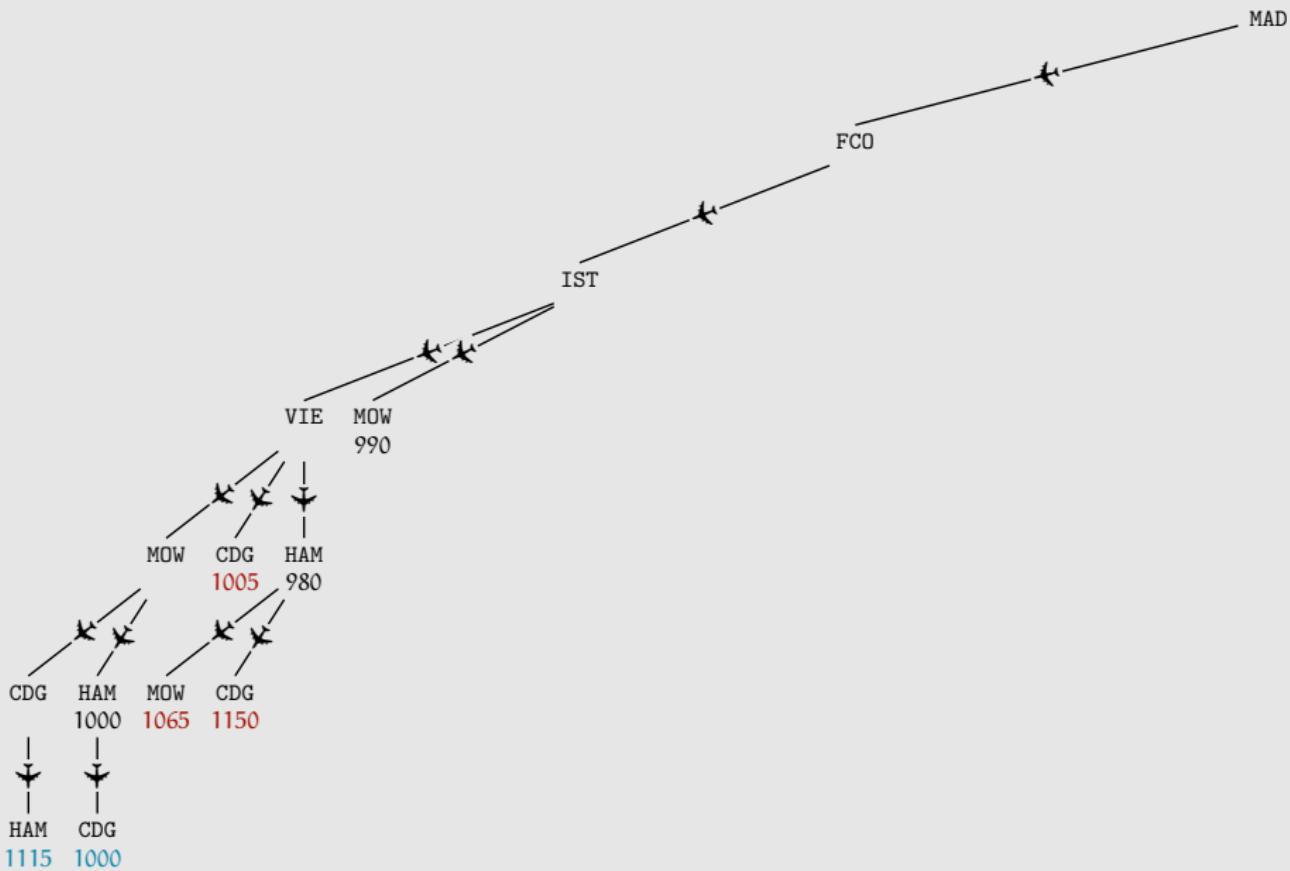
Problema del viajante:

branch & bound



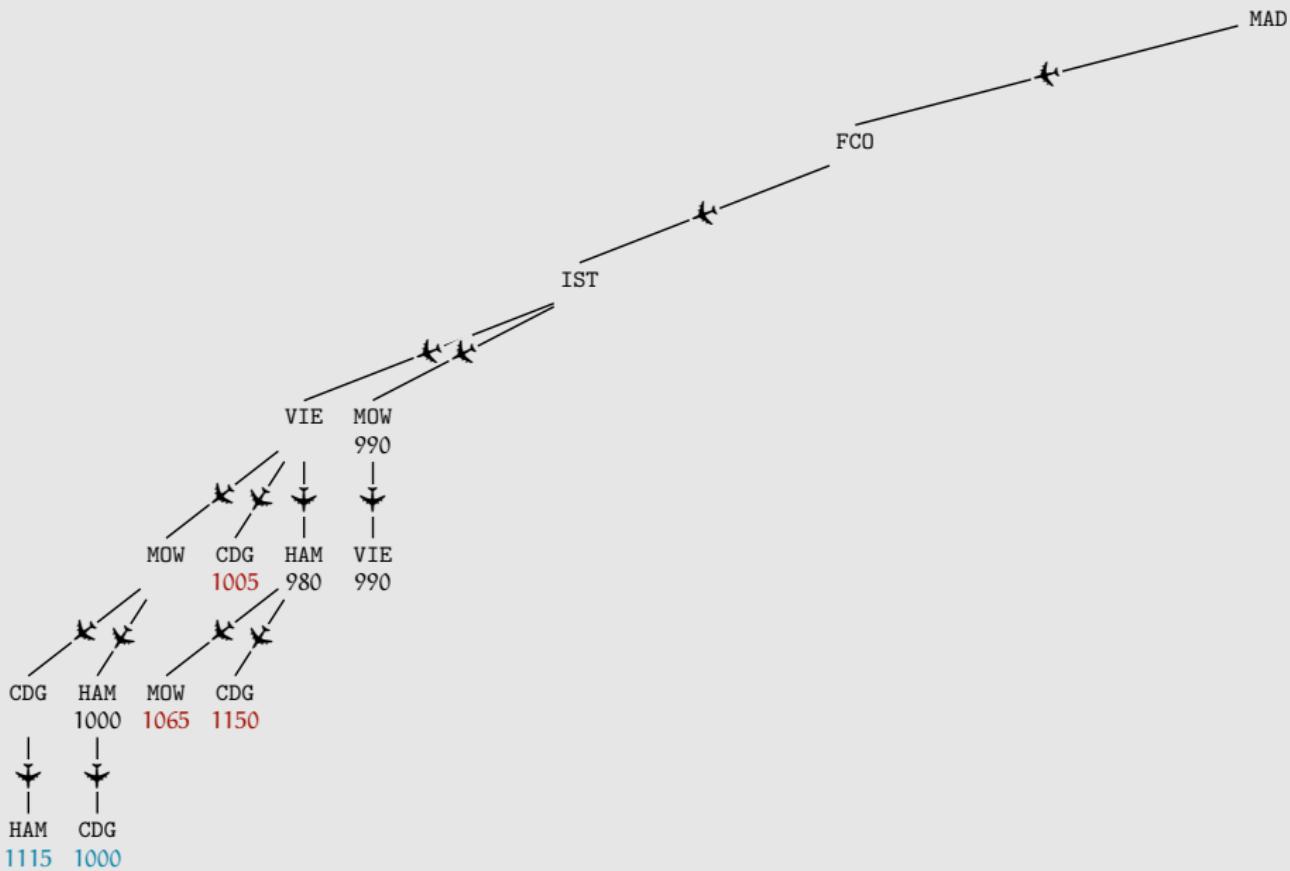
Problema del viajante:

branch & bound



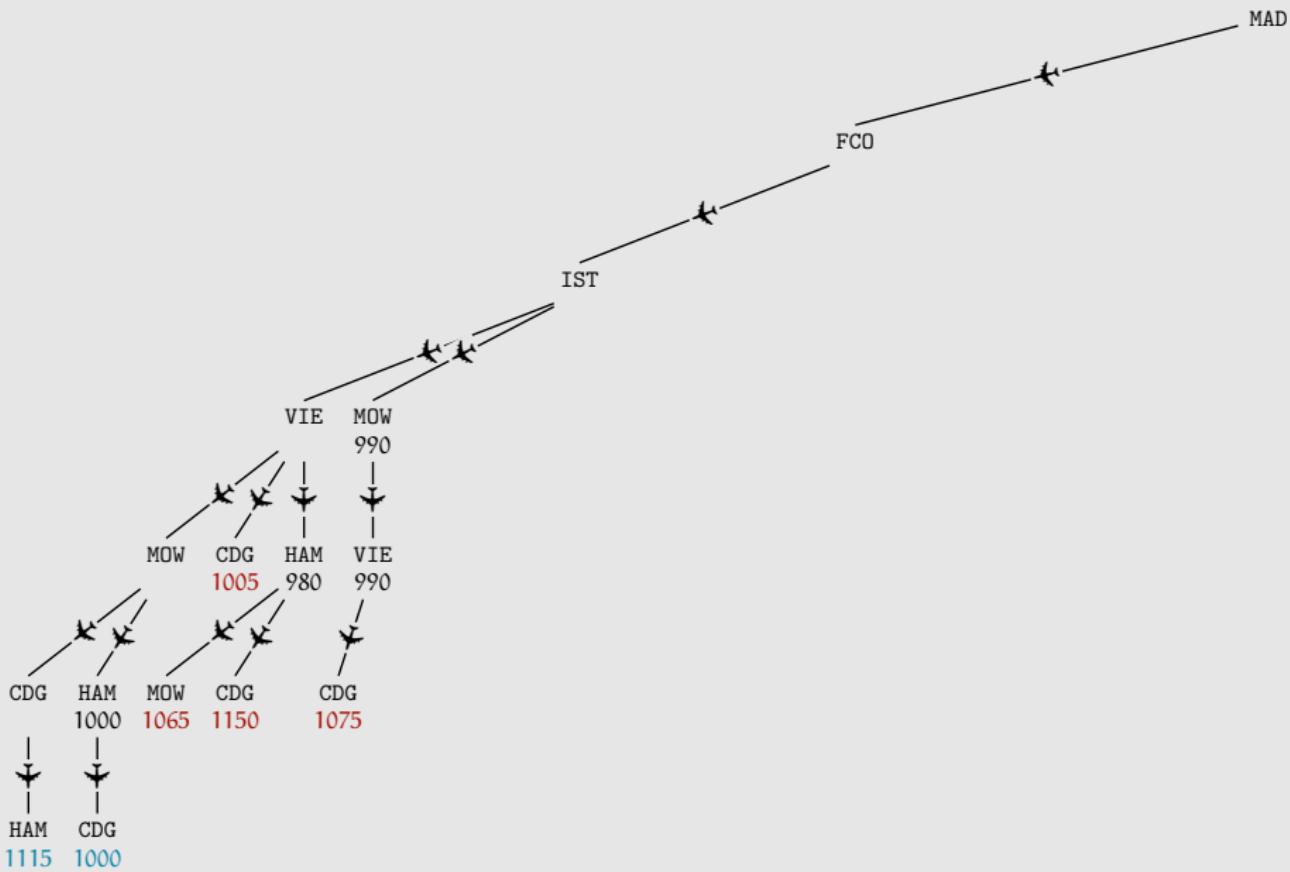
Problema del viajante:

branch & bound



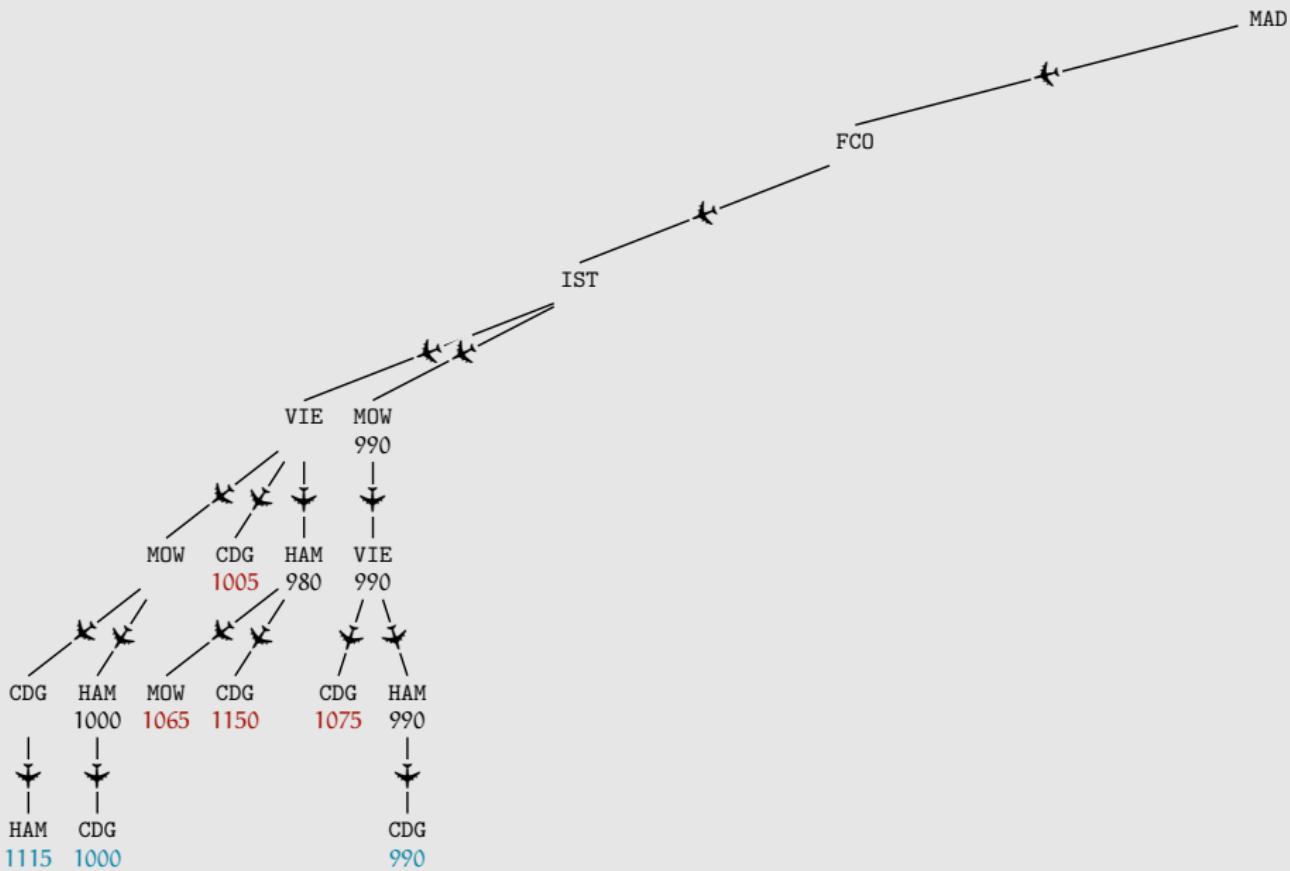
Problema del viajante:

branch & bound



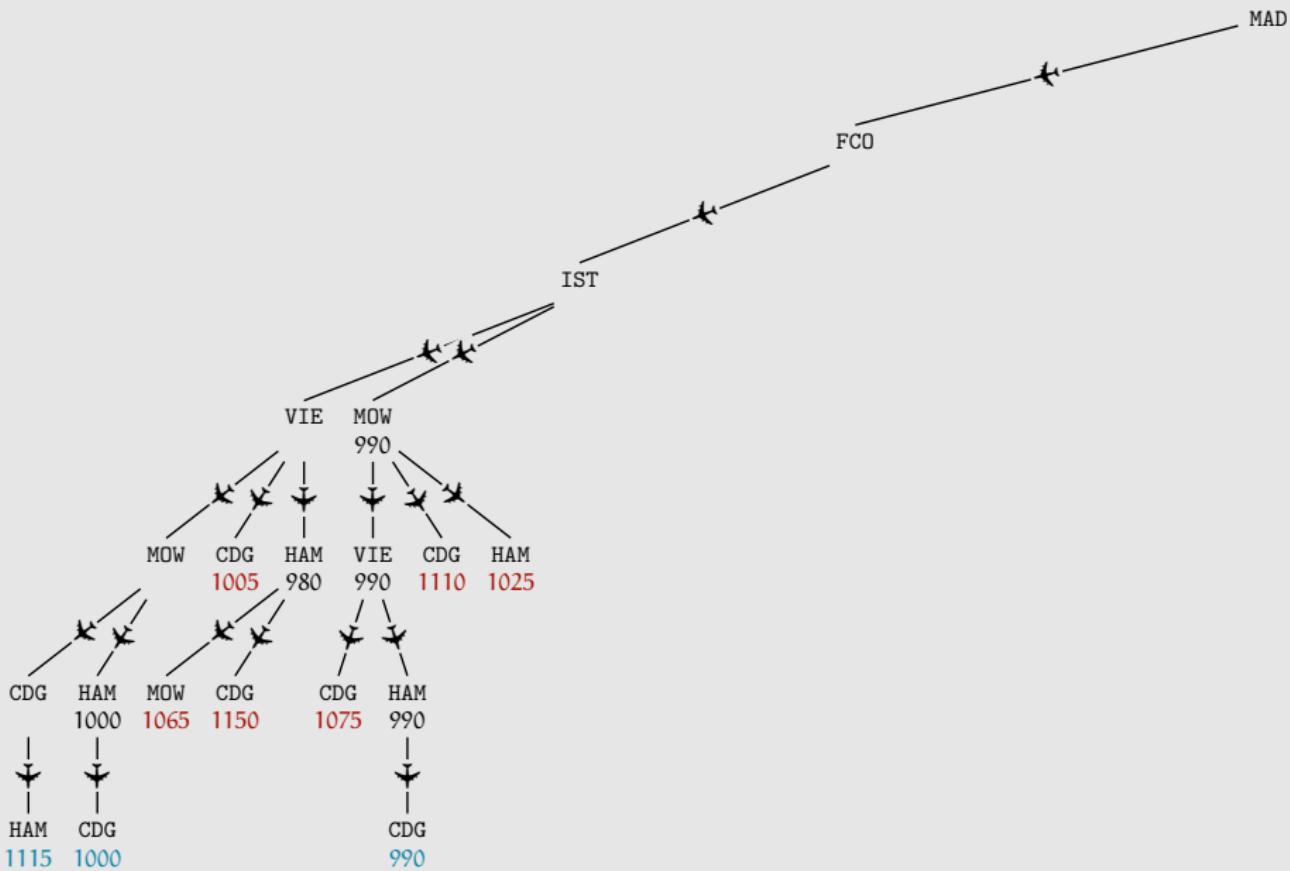
Problema del viajante:

branch & bound



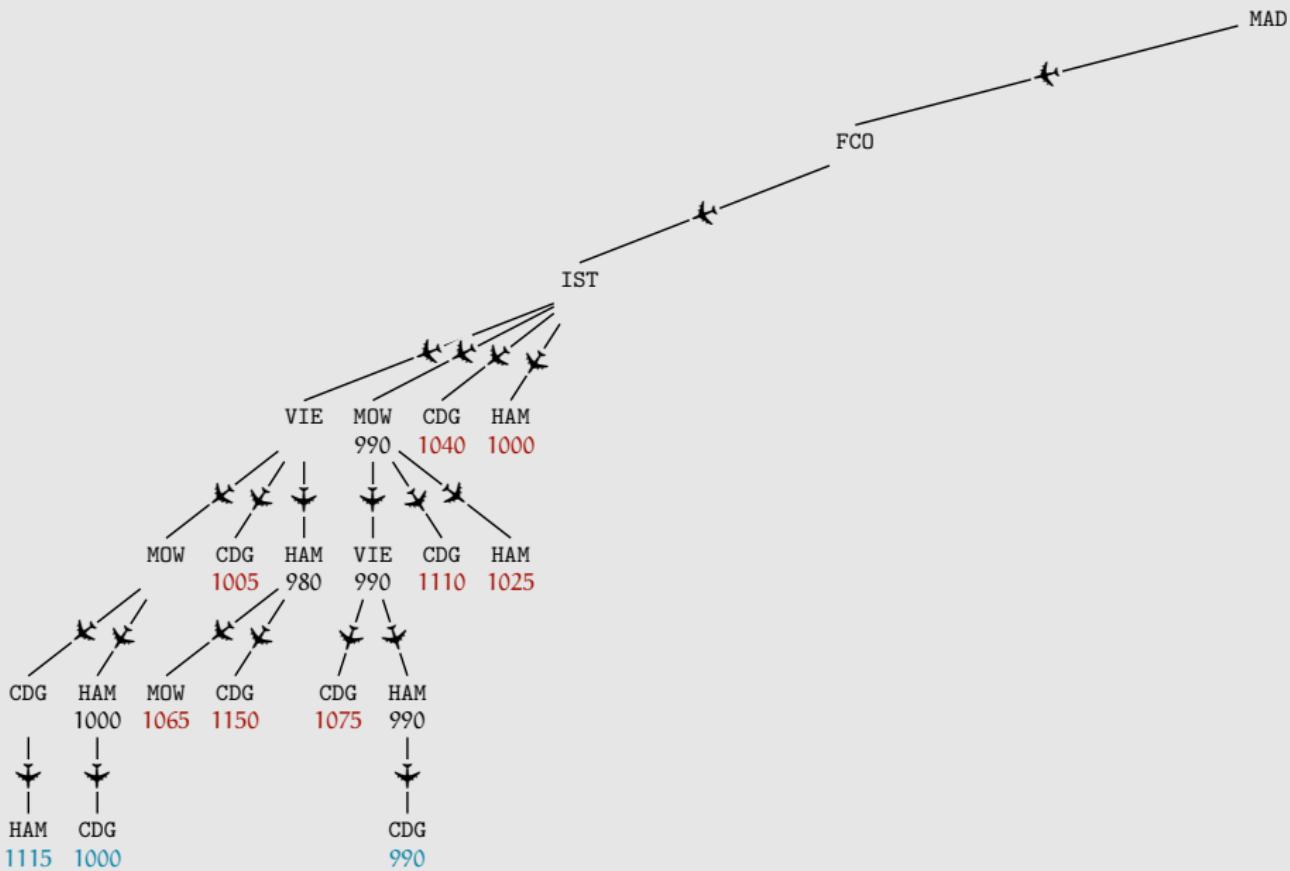
Problema del viajante:

branch & bound



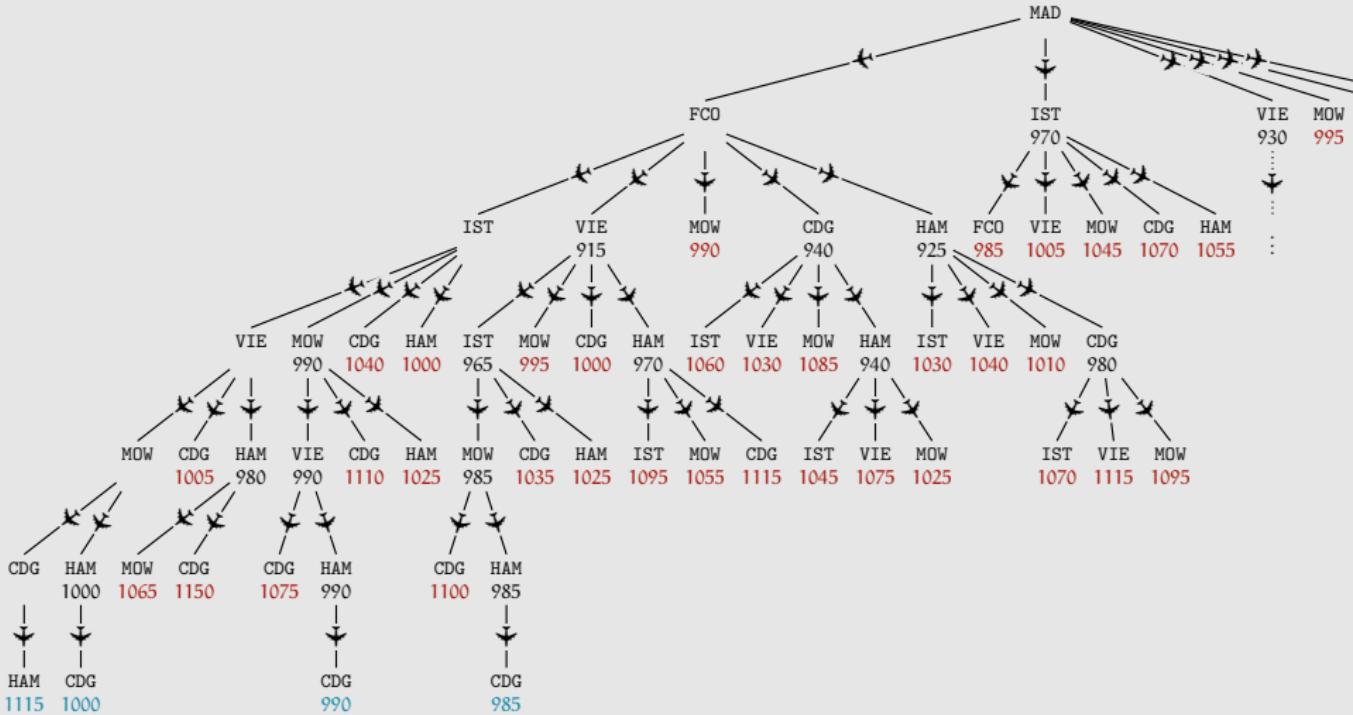
Problema del viajante:

branch & bound



Problema del viajante:

branch & bound



dia_00_cod_22.py

```
def lanza(p, mj=None, t_mj=infinito):
    ct = p.cota()
    fondo = p.es_completa()
    escribe_línea(p, ct, t_mj, fondo, mejora)
    if ct < t_mj:
        if fondo:
            mj = p
            t_mj = ct
        else:
            for ap in p.amplía():
                mj, t_mj = lanza(ap, mj, t_mj)
    return mj, t_mj
```
