Halting Problem (HP)

Comparación de métodos exactos, heurísticos y aproximados

Diego Acuña – TC – Marzo 2026

Bibliografía

- https://cs.uns.edu.ar/materias/tc2do/downloads/Apuntes/Funciones%20Recursivas-TdC2019.pdf

Ejemplo 1: ¿Termina para toda entrada?

```
int f (int x, int y) {
    return x + y
}
```

Ejemplo 2: ¿Termina para toda entrada?

```
int f (int x, int y) {
    return x / y
}
```

Ejemplo 3: ¿Termina para toda entrada?

```
int f (int x, int y) {
    while (true){}
    return 0;
}
```

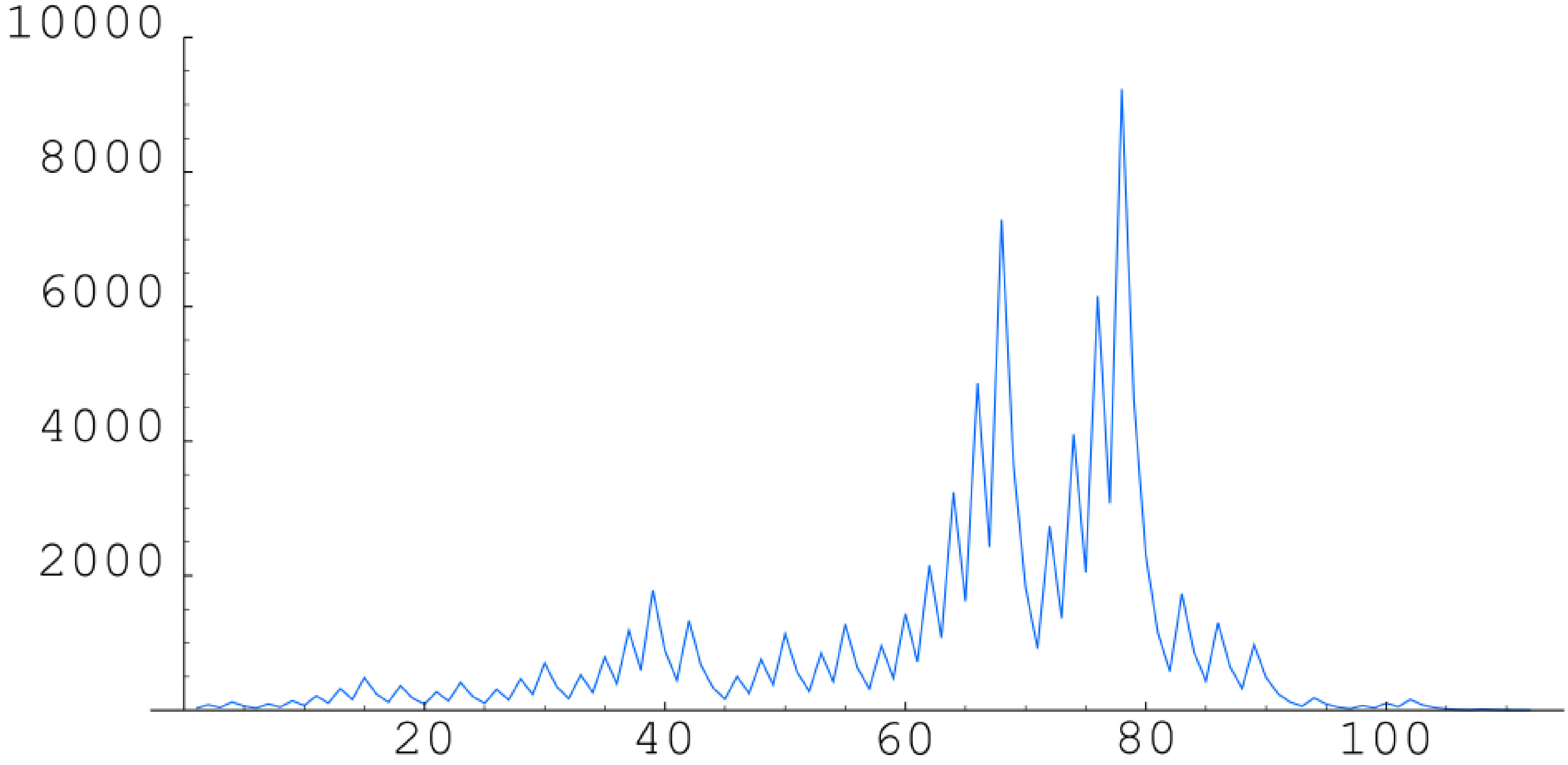
Ejemplo 4: ¿Termina para toda entrada?

```
int f (int n) {
    if ( n == 0 ) {
        return  f(0);
    } else {
        return 0;
    }
}
```

Ejemplo 5: ¿Termina para toda entrada?

```
int f (int n) {
    if ( n == 1 ) {
        return  1;
    } else {
        if (n % 2  == 0 ) { return f(n/2) }
        else { return f(3*n+1) }
    }
}
```

Conjetura de Collatz



¿Terminan? ¿Qué tienen en común?

int **suma**(int a, int b) {
  if (b == 0)
    return a;
  else
    return suma(a, b - 1) + 1;}

int **factorial**(int n) {
  if (n == 0)
    return 1;
  else
    return multiplicacion(n, factorial(n - 1));}

int **minimo**(int a, int b) {
  if (a == 0 || b == 0) return 0;
  else return 1 + minimo(a - 1, b - 1);}

Recursión primitiva

- $\boxed{\text{Recursión primitiva}}$: Sea g una función recursiva primitiva total n-aria, y sea h una función recursiva primitiva total $n+2$-aria. Luego puede definirse una función f, $n+1$-aria, tal que para toda n-upla $(x_1, \ldots, x_n) \in \mathsf{Nat}^n$ y $m \in \mathsf{Nat}$ se tiene:

$$\begin{array}{rcl} f(x_1, \ldots, x_n, 0) & = & g(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, m+1) & = & h(x_1, \ldots, x_n, m, f(x_1, \ldots, x_n, m)) \end{array}$$

En este caso se dice que f se obtiene a partir de g y h por *recursión primitiva*.

Algoritmo ack:
Datos de entrada: a_1, a_2
Datos de salida: $Resultado$
Comienzo
  Si $a_1 = 0$
    entonces
      $Resultado \leftarrow a_2 + 1$
    si no
      si $a_2 = 0$
        entonces
          $Resultado \leftarrow ack(a_1 - 1, 1)$
        si no
          $Resultado \leftarrow ack(a_1 - 1, ack(a_1, a_2 - 1))$
      fin si
  fin si.
Fin Algoritmo

$$ack(0, a_2) = a_2 + 1$$
$$ack(a_1 + 1, 0) = ack(a_1, 1)$$
$$ack(a_1 + 1, a_2 + 1) = ack(a_1, ack(a_1 + 1, a_2))$$

Recursión estructural: Árboles

- Ej: Altura, cantidad de nodos, máximo, etc...
- Toda función definida por recursión estructural sobre un tipo inductivo finito **siempre termina**.

Motivo:

- Existe una relación de orden bien fundado ($\prec$) sobre las subestructuras
- Cada llamada recursiva reduce estrictamente la medida estructural
- No existen cadenas infinitas descendentes

Recursión bien fundada

Ejemplo: ¿Termina?

sumIntervalo desde hasta = $\sum_{i=desde}^{hasta} i$

```
sumIntervalo :: Int -> Int -> Int
sumIntervalo desde hasta
 | desde <= hasta  = desde + (sumIntervalo (desde+1) hasta
 | otherwise = 0
```

Recursión bien fundada

Sea $(D, \prec)$un conjunto con una relación **bien fundada** $\prec$(es decir, sin cadenas infinitas descendentes).

Definición formal:

$$f(x) = F(x,\ \{f(y) \mid y \prec x\})$$

Una función está definida por **recursión bien fundada** si cada llamada recursiva se realiza **únicamente sobre elementos estrictamente menores** que el argumento actual respecto a una relación bien fundada $\prec$.

Recursión

$$\boxed{\text{Primitiva} \subset \textit{Estructural} \subset \textit{Bien fundada}}$$

- Toda recursión **primitiva** es estructural sobre $\mathbb{N}$
- Toda recursión **estructural** es bien fundada
- No toda recursión bien fundada es estructural
- No toda recursión estructural es primitiva

Preguntas

- “¿Cómo distinguirías entre un programa que no termina y uno que tarda mucho?”
- “¿Qué significa saber con certeza que no termina?”
- “¿Puedes demostrar que no va a terminar sin ejecutarlo?”
- “¿Qué pasa si el programa se analiza a sí mismo?”
- “¿Puede un programa predecirse a sí mismo?”

Observaciones

- Podemos detectar la terminación en algunos casos particulares, pero el desafío es decidirla para todos los programas posibles.
- Un timeout no resuelve el problema: si un programa no terminó aún, no sabemos si va a terminar más adelante o si nunca lo hará.
- La dificultad surge porque los programas pueden contener loops y recursión sin límite, lo que genera comportamientos potencialmente infinitos.
- Si restringimos el lenguaje (por ejemplo, eliminando loops no acotados), ahí podríamos.

Observaciones

Pero entonces qué hacen...

- Los antivirus
- Los compiladores
- El Sistema operativo
- El debugger

Algunas conclusiones

- No puede existir un compilador perfecto que detecte todos los loops infinitos posibles. DEMOSTRACIÓN
- Ningún algoritmo —ni siquiera una IA— puede resolver el problema en general, porque es una limitación matemática demostrada.

Ejemplo Dafny

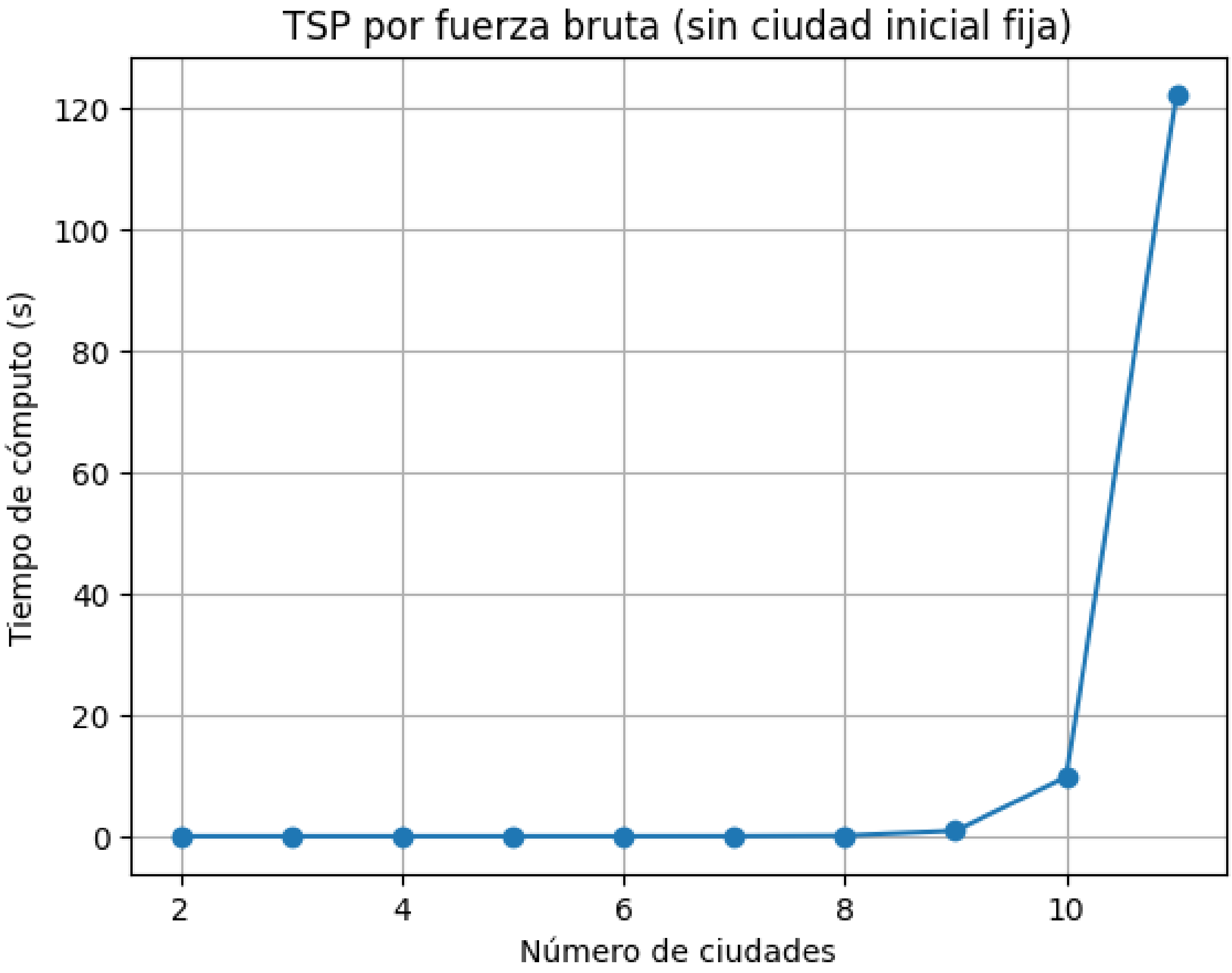
```
method factorial1(n:nat) returns (b:nat)
{
if n==0 {return 1;}
var aux:= factorial1(n-1);
b:= n*aux;
}

method factorial2(n:nat) returns (b:nat)
{
if n==0 {return 1;}
var aux := factorial2(n);    cannot prove termination; try supplying a decreases clause
b:= n*aux;
}       You, 2 weeks ago • nats and lists
```

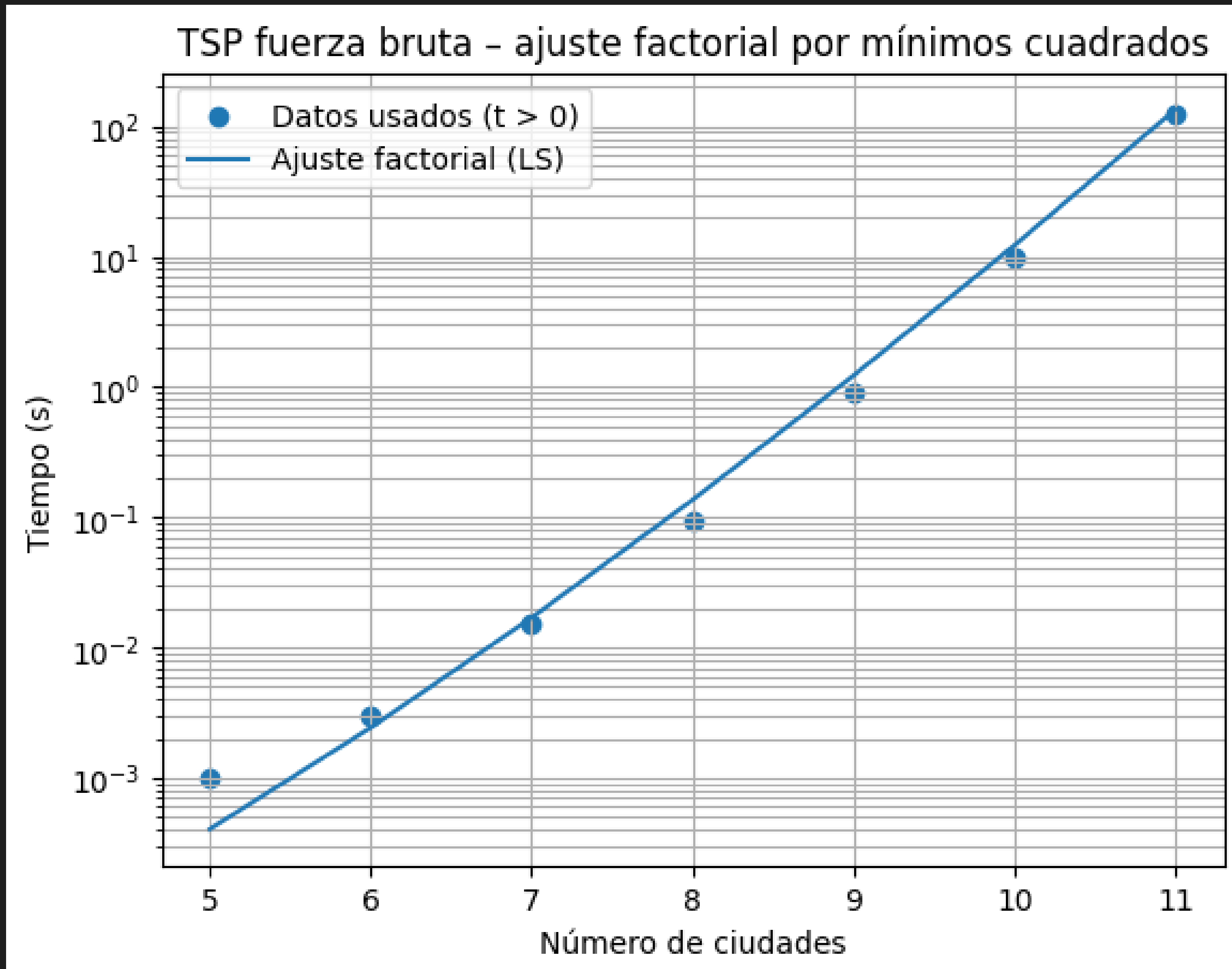
Traveling Salesman Problem (TSP)

Comparación de métodos exactos, heurísticos y aproximados

Fuerza Bruta – Tiempo vs n

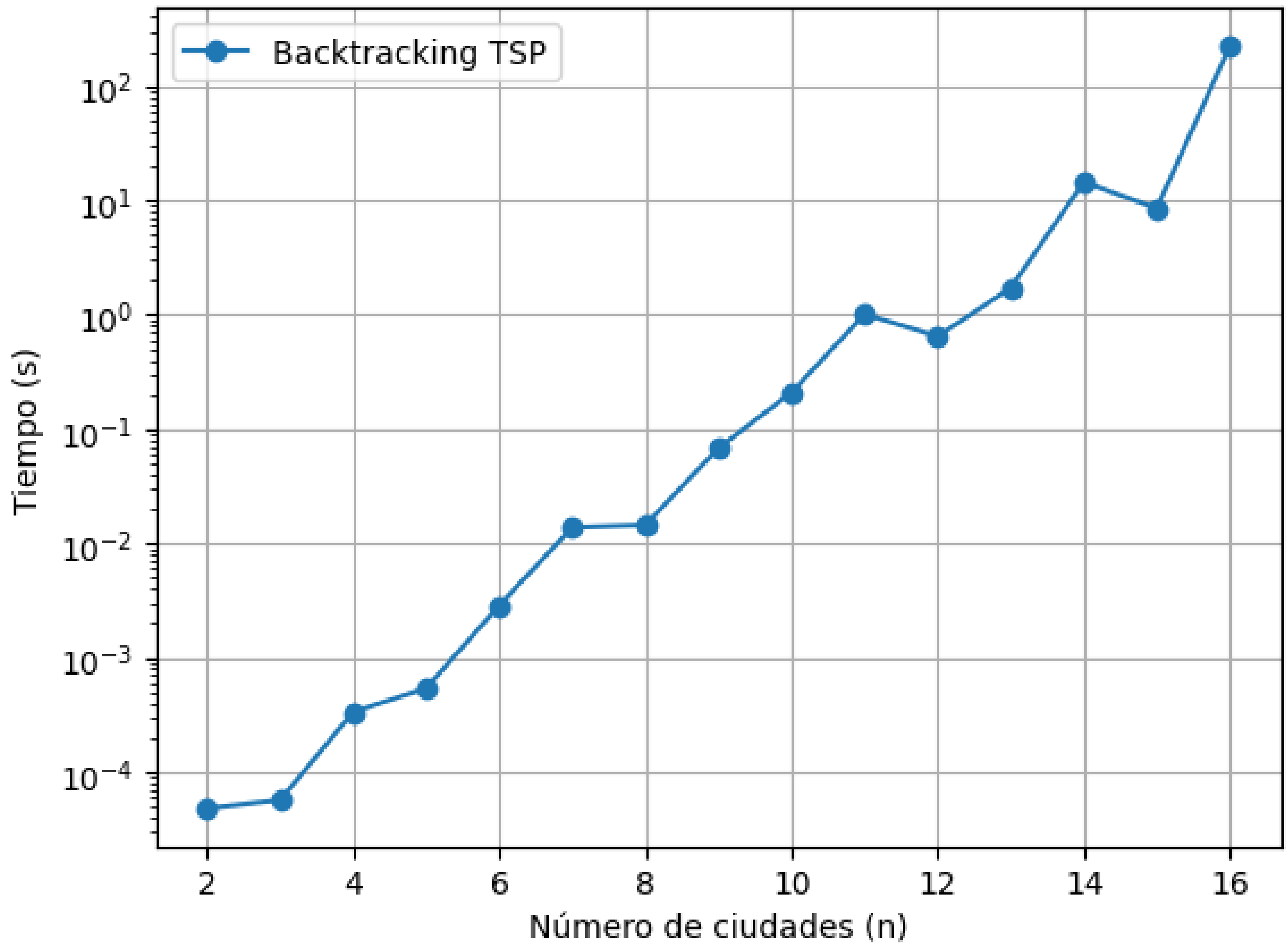


```
Ajuste factorial (mínimos cuadrados)
c = 3.361e-06
Modelo: t(n) = c * n!
```

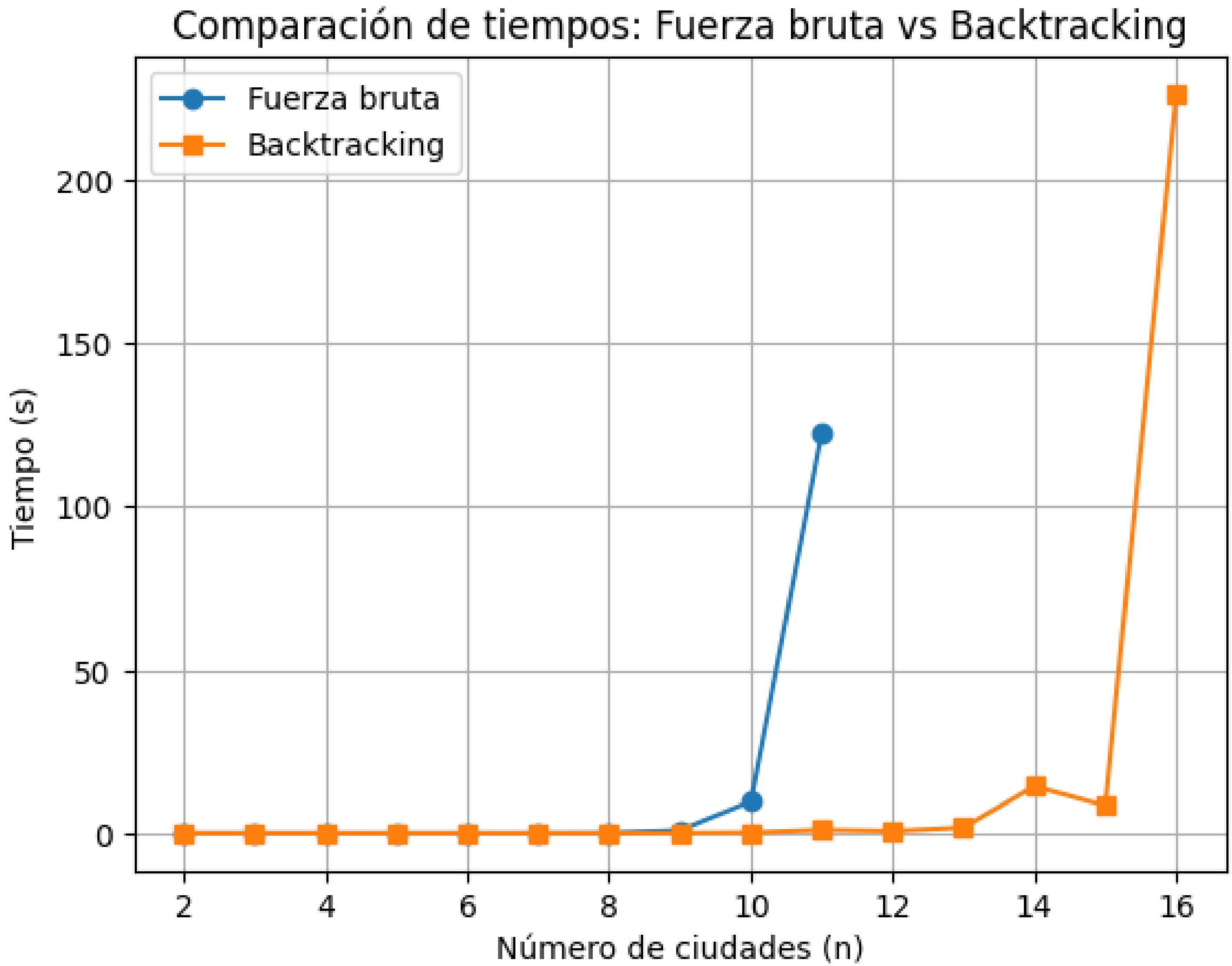


Tiempo estimado para n=19:
1.296 años (en mi pc)
6.758.905 años (en Google Colab)

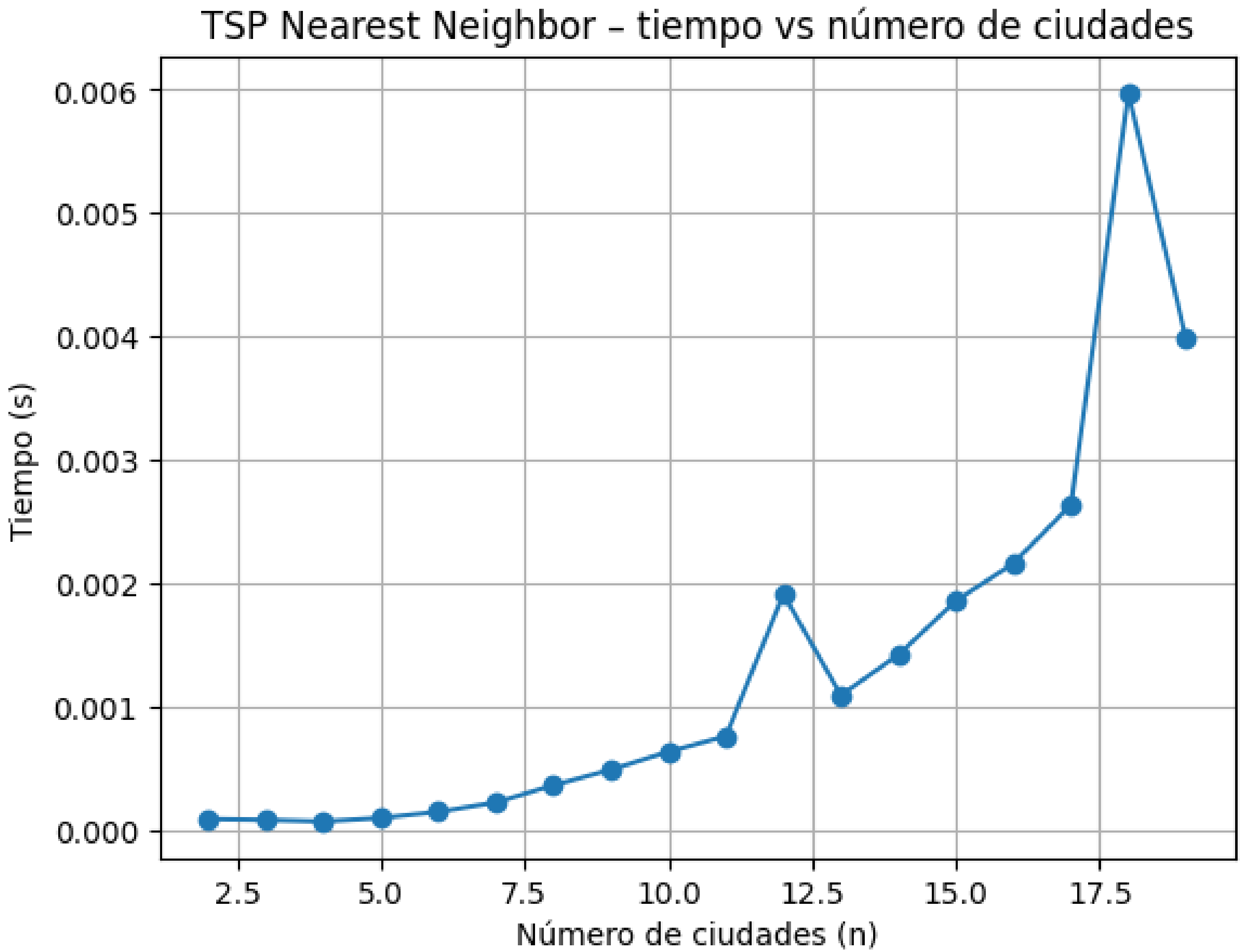
Backtracking



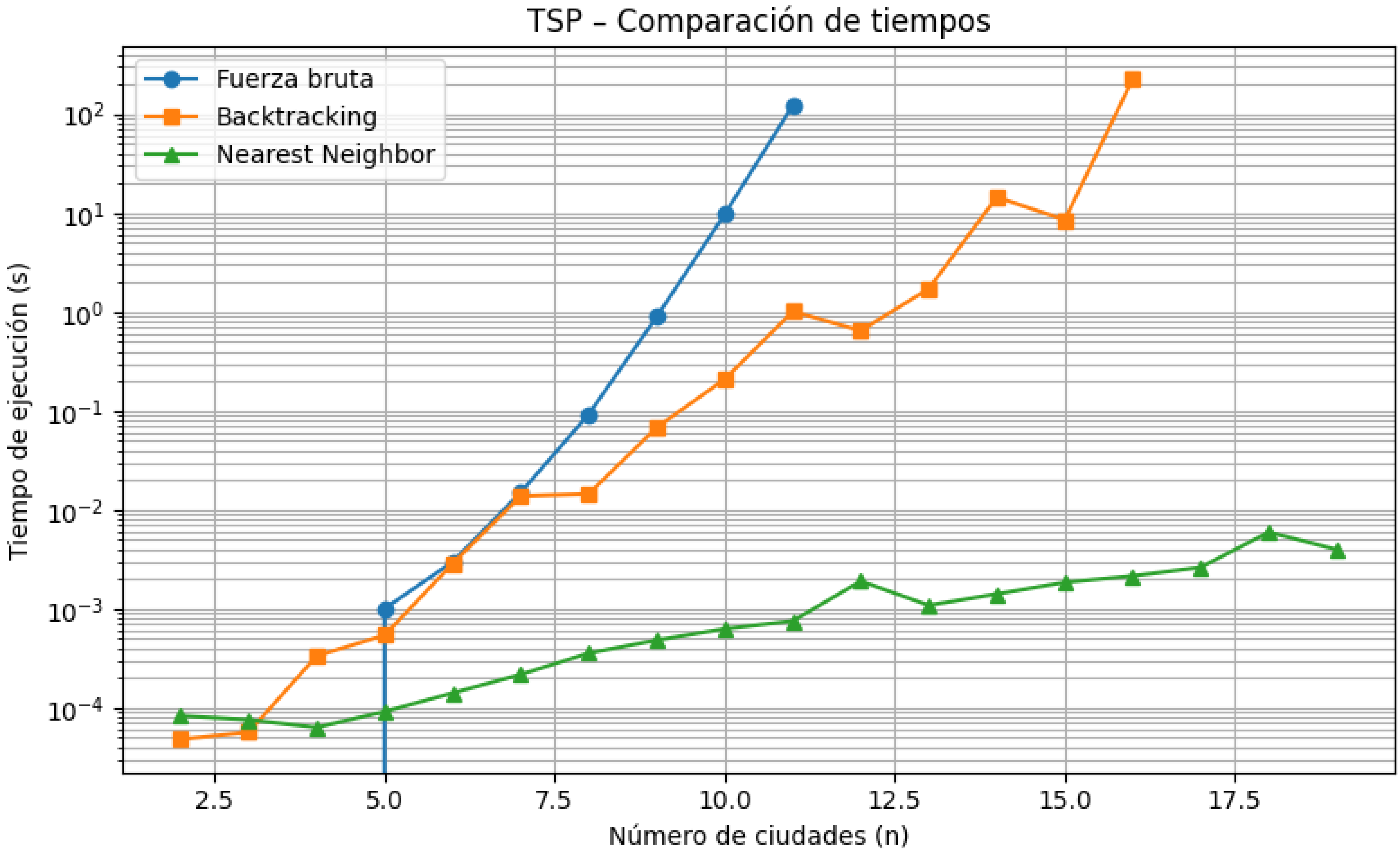
BF vs Backtracking



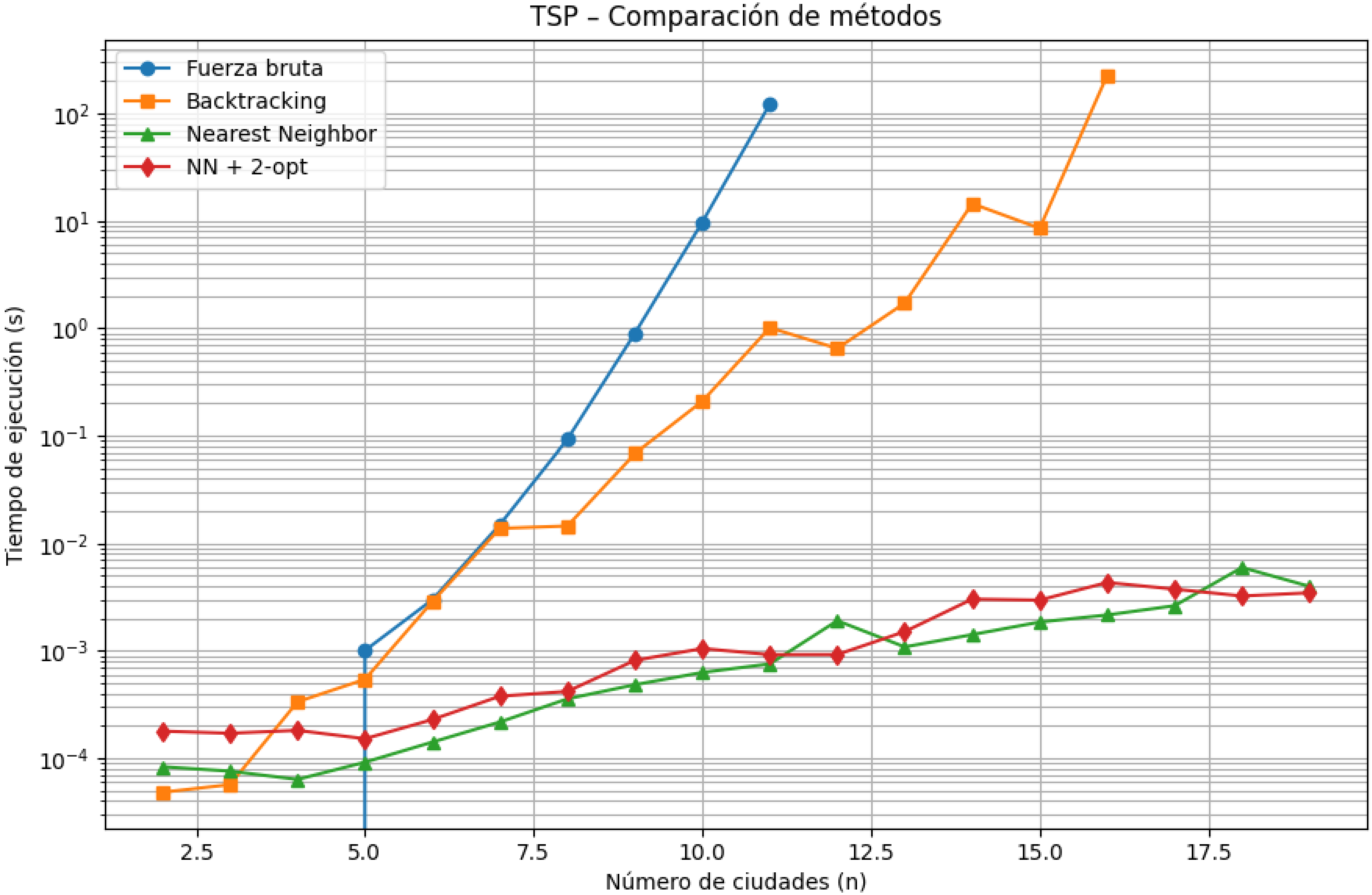
Greedy – Nearest Neighbor



Comparación de los 3



NN+2-Opt + el resto



Método	Orden de complejidad	¿Exacto?	¿Cota superior de error?
Fuerza Bruta (FB)	(O(n!))	✅ Sí	0 (óptimo garantizado)
Backtracking (BT)	Exponencial (peor caso (O(n!)))	✅ Sí	0 (óptimo garantizado)
Nearest Neighbor (NN)	(O(n^3)) *(todos los starts)*	❌ No	❌ No (puede ser muy malo)
NN + 2-opt	(O(n^3))–(O(n^4))	❌ No	❌ No (mejora empírica)

Método	Exacto	Garantía	Cota
Fuerza Bruta	✅	Sí	1.0
Backtracking	✅	Sí	1.0
Double Tree	❌	Sí	≤ 2
Christofides	❌	Sí	≤ 1.5
Nearest Neighbor	❌	No	—
NN + 2-opt	❌	No	—

Double Tree

Double Tree construye primero un árbol generador mínimo (MST) que conecta todas las ciudades con costo mínimo. Luego duplica sus aristas para poder recorrerlas y genera un tour "atajando" ciudades ya visitadas. Es un algoritmo aproximado de orden (O(n^2)) que garantiza una solución de costo a lo sumo el doble del óptimo en TSP métricos.

Christofides

Christofides parte de un árbol generador mínimo y agrega un matching mínimo entre los nodos de grado impar. Con esto construye un recorrido euleriano que se transforma en un tour hamiltoniano. Es un algoritmo aproximado de orden (O(n^3)) que garantiza una solución de costo a lo sumo 1.5 veces el óptimo en TSP métricos.