UNIVERSIDAD DE GRANADA

Algoritmos voraces (*greedy*)

Algorítmica. Práctica 3

Celia Arias Martínez
Miguel Ángel Fernández Gutiérrez
Sergio Quijano Rey
Lucía Salamanca López

segfault

Contenidos

Introducción

Objetivo

Apreciar la utilidad de los métodos voraces (*greedy*) para resolver problemas de forma muy eficiente, en algunos obteniendo soluciones óptimas y en otros aproximaciones. Para ello realizamos:

- **Problema común:** problema del viajante de comercio (TSP).
- **Problema asignado:** trabajadores y tareas (*worker*).

Problema del viajante de comercio (TSP)

Problema común

Travelling Salesman Problem (TSP)

Dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la distancia recorrida sea mínima.

Formalmente: dado un grafo G, conexo y ponderado, se trata de hallar el *ciclo hamiltoniano* de mínimo peso de ese grafo.

Enfoque 1: por cercanía

Dada una ciudad inicial v_0, se agrega como ciudad siguiente aquella v_i (no incluída en el circuito) que se encuentre más cercana a v_0. El procedimiento se repite hasta que todas las ciudades se hayan visitado.

Enfoque 1: por cercanía

PASOS

1. Partimos del nodo 0
2. Encontramos, en el vector de nodos disponibles, el más cercano al anterior.
3. Eliminamos el escogido del vector.
4. Repetimos los pasos anteriores con los nuevos nodos hasta que el vector se quede vacío.
5. Finalmente devuelve el camino.

Enfoque 1: por cercanía

Para este enfoque usamos:

- Un *struct* `Point`, que tiene una coordenada `x`, una coordenada `y`, y una función `distancia` para calcular la distancia entre dos `Point`.
- Una función `get_best_solution`, que calcula la solución especificada anteriormente. Para ello, hace uso de:
  - `road`, un vector donde se almacenan las soluciones parciales, es decir, las que resultan de añadir un nodo al recorrido.
  - `points_left`, vector donde almacenamos los nodos que nos quedan por recorrer.

Enfoque 1: por cercanía

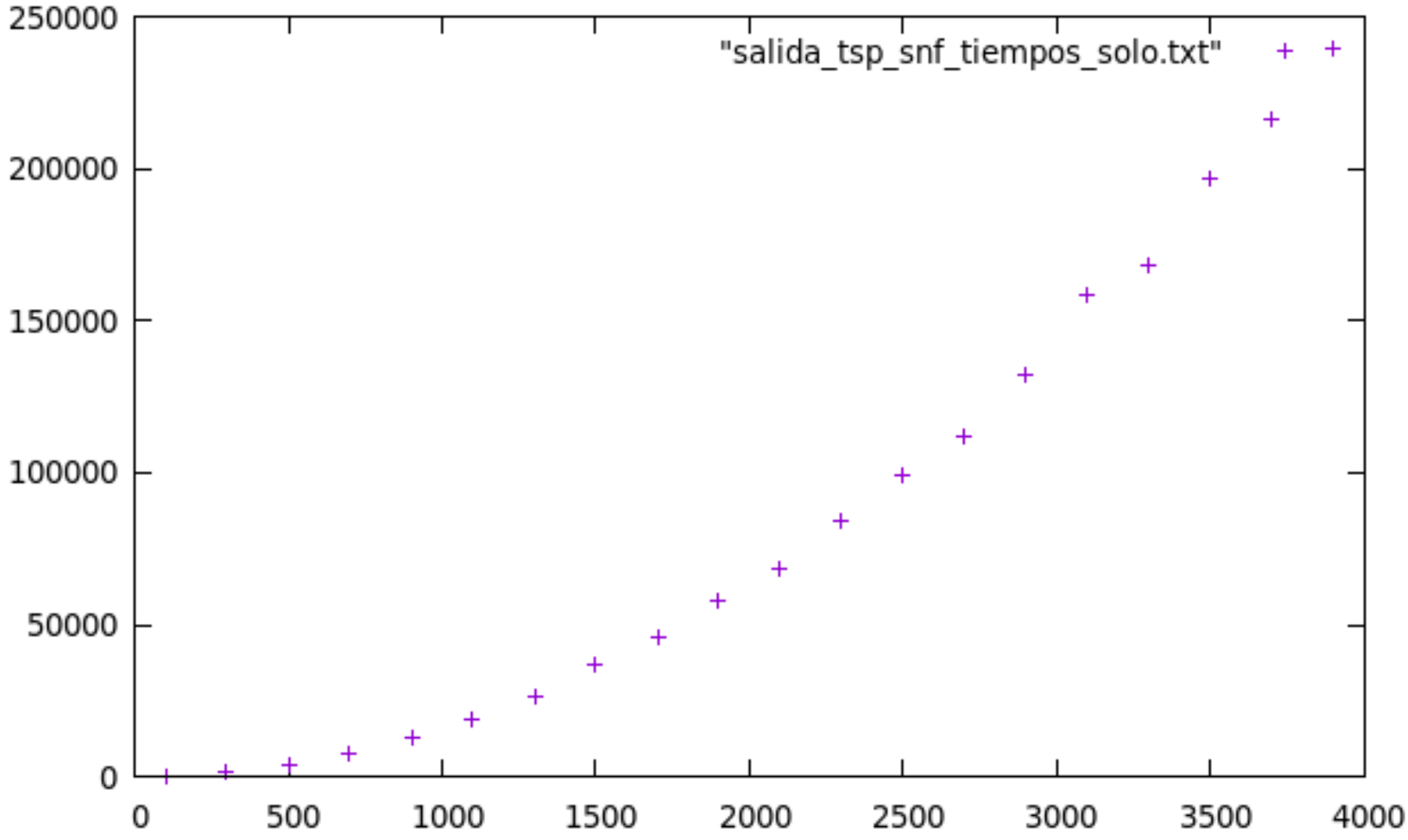
Guardamos en `points_left` todos los nodos y en `road` el primer punto, que podemos asumir que es el primero.

Mientras que el vector `points_left` no esté vacío calculamos la distancia de todos esos nodos al último Punto de `road` y añadimos el nodo que esté a la menor distancia, borrándolo de `points_left`.

Enfoque 1: por cercanía. Análisis empírico

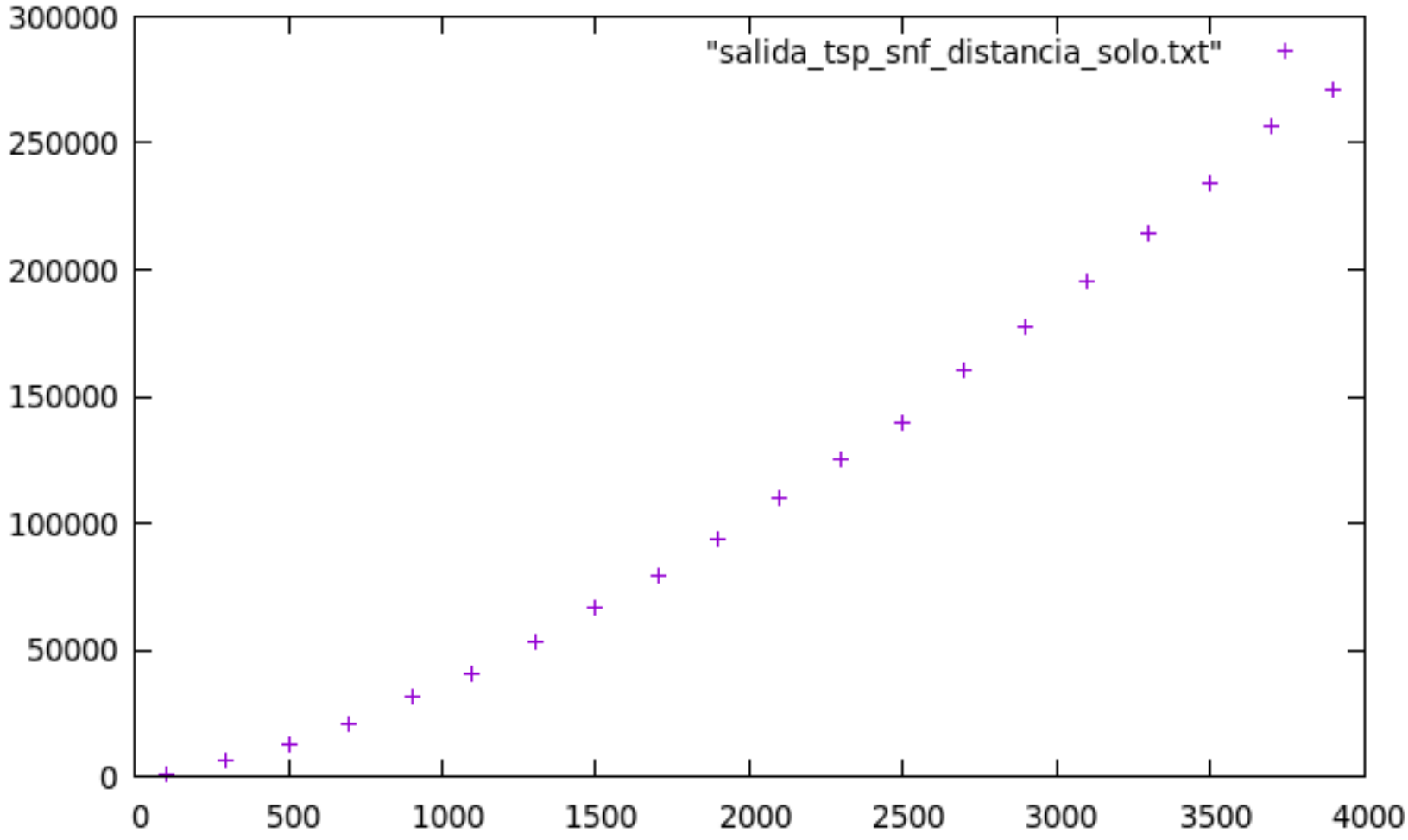
- **Tamaños de prueba:** desde 100 hasta 4000 ciudades, incremento de 200.
- Cada iteración la repetimos 100 veces y hacemos la media, con el fin de eliminar peores y mejores casos.

Enfoque 1: por cercanía. Análisis empírico



Datos empíricos para viajante de comercio versión cercanía, tiempo

Enfoque 1: por cercanía. Análisis empírico



Datos empíricos para viajante de comercio versión cercanía, distancia

Enfoque 2: por inserción

Se comienza con un recorrido parcial y luego se extiende insertando las ciudades restantes mediante algún criterio de tipo voraz: insertando los nodos de modo que el recorrido sea mínimo.

Enfoque 2: por inserción

PASOS

1. Partimos de tres nodos:
   - El que está más al *Norte*.
   - El que está más al *Este*.
   - El que está más al *Oeste*.
2. Partiendo de este camino añadimos el punto de forma que la distancia del nuevo camino sea mínima.
3. Repetimos esto sucesivamente hasta que tengamos todos los puntos.
4. Finalmente devuelve el camino.

Enfoque 2: por inserción

Hacemos uso de:

- `candidates` es el vector con los nodos que faltan por insertar.
- `road` es el vector con la solución parcial.
- `most_north` es el nodo más al norte.
- `most_west` es el nodo más al oeste.
- `most_east` es el nodo más al este.
- `get_best_candidate` es una función auxiliar que dado un camino y un vector de candidatos devuelve un vector con la posición del punto óptimo y dónde queremos insertarlo en el camino.

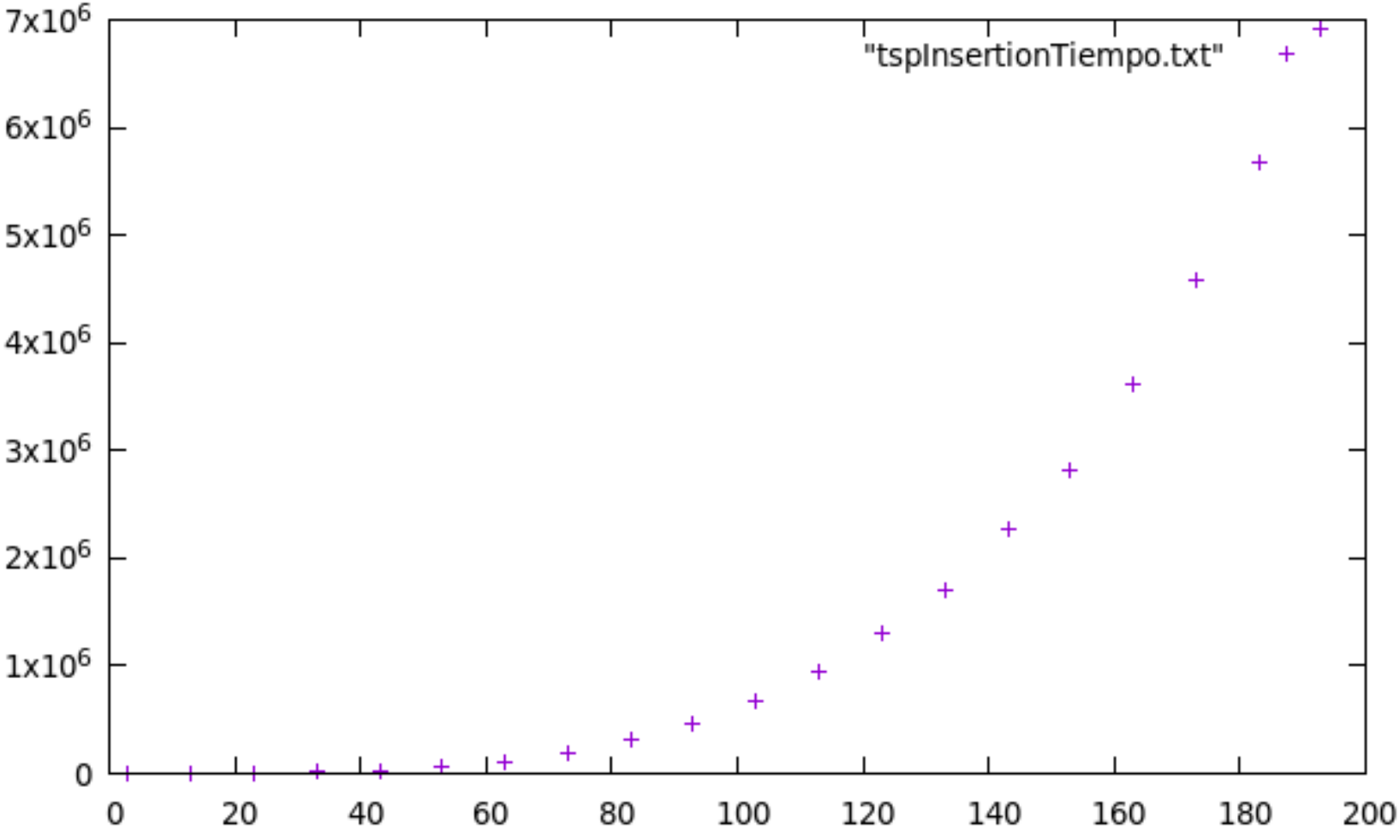
Enfoque 2: por inserción

En la nueva función `get_best_solution`, primero calculamos e insertamos en `road` los nodos más al norte, este y oeste, para que se queden dentro los máximos nodos posibles. Mientras que `candidates` no sea vacío calculamos el nodo y la posición con los cuales la distancia que tenemos que recorrer aumenta lo mínimo posible al añadir un nodo, y lo insertamos en `road`, quitándolo de `candidates`.

Enfoque 2: por inserción. Análisis empírico

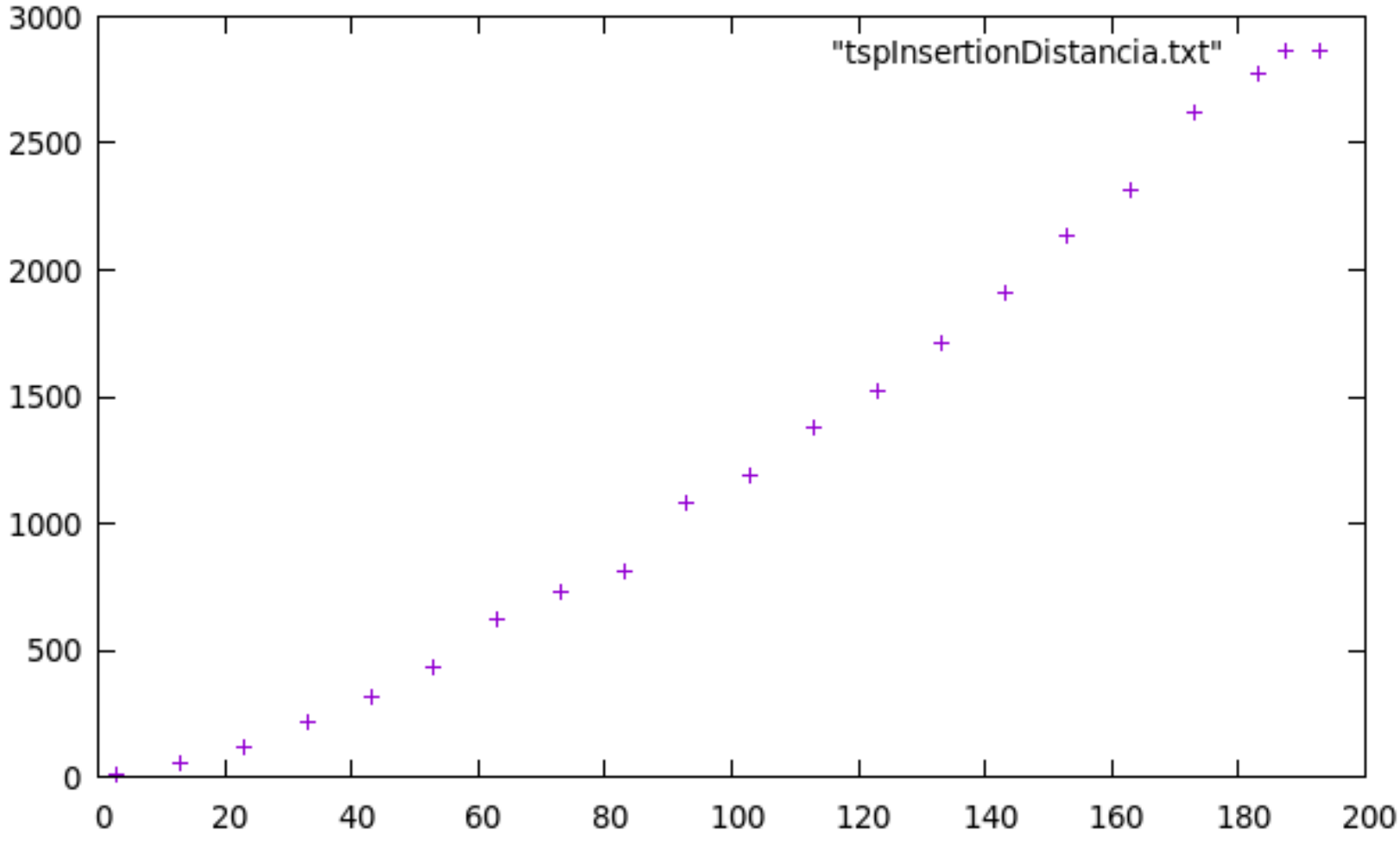
- **Tamaños de prueba:** desde 3 hasta 200 ciudades, con incremento de 10.
- Cada iteración la repetimos 100 veces y hacemos la media, con el fin de eliminar peores y mejores casos.

Enfoque 2: por inserción. Análisis empírico



Datos empíricos para viajante de comercio versión inserción, tiempo

Enfoque 2: por inserción. Análisis empírico



Datos empíricos para viajante de comercio versión inserción, distancia

Enfoque 3: por perturbaciones

Este enfoque, de nuevo *greedy*, dado un recorrido, realiza las perturbaciones indicadas por un parámetro para intentar mejorarlo.

Enfoque 3: por perturbaciones

PASOS

1. Calculamos la solución dada por el algoritmo SNF.
2. Calculamos el nodo respecto el que tenemos que perturbar.
3. Perturbamos y vemos la ganancia del camino.
4. Si la ganancia es mejor se cambia el camino.
5. Se devuelve el camino.

Enfoque 3: por perturbaciones

Hacemos uso de:

- `points` es un vector con los nodos dados.
- `perturbations` es el número de perturbaciones que aplicamos al algoritmo.
- `get_snf_solution` es una función auxiliar que calcula según el algoritmo por cercanía una solución al conjunto de puntos.
- `get_worst_node` es una función auxiliar que calcula el nodo tal que según el recorrido actual su distancia al siguiente punto es la mayor.
- `perturbate` es una función auxiliar que encuentra un camino diferente que haga que el peor nodo mejore y sobrescribe el camino actual.

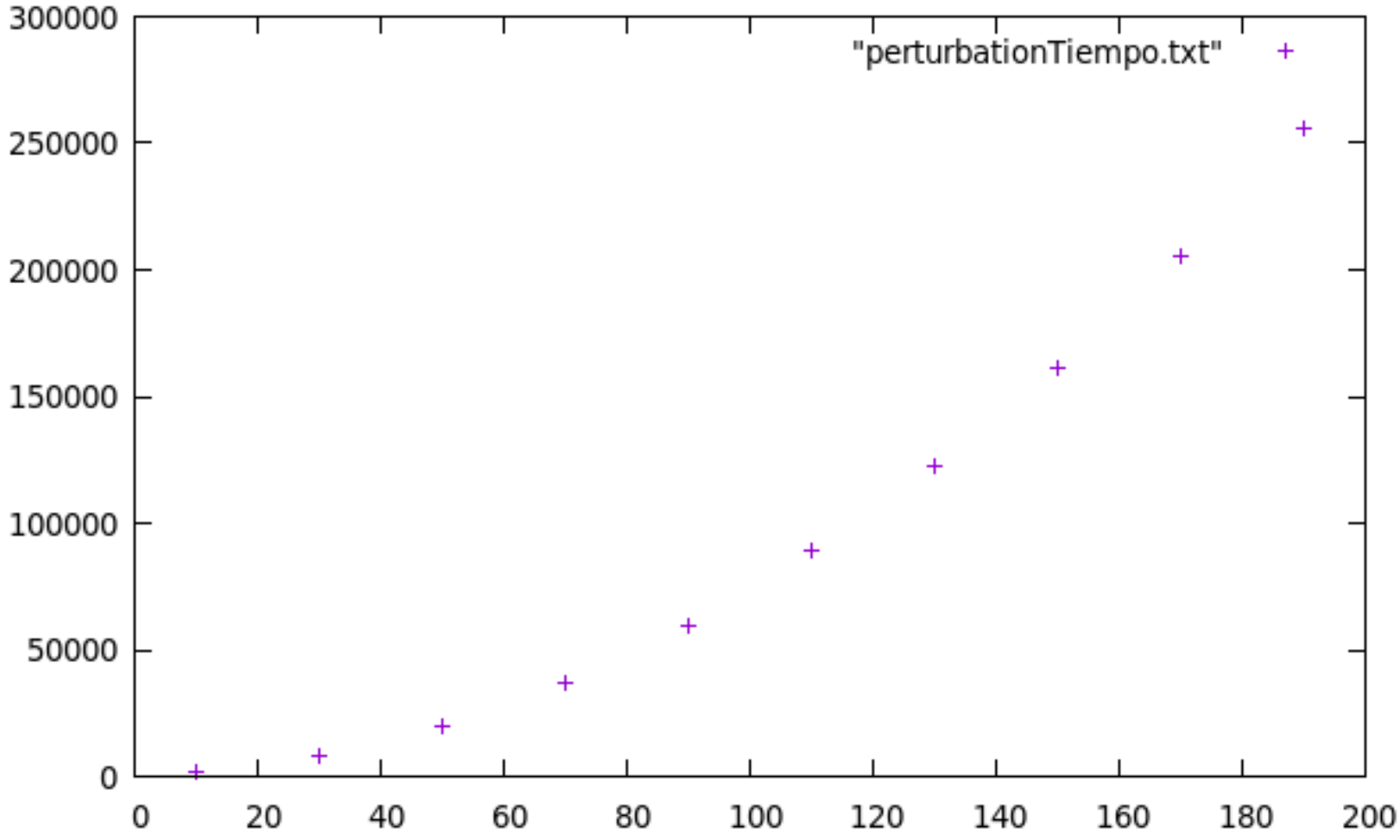
Enfoque 3: por perturbaciones

Primero calculamos una solución inicial con el algoritmo de *cercanía*. Después calculamos el peor nodo de ese recorrido e intentamos encontrar otra combinación de nodos que mejore ese nodo en concreto. Este proceso lo repetimos tantas veces como `perturbations` indique.

Enfoque 3: por perturbaciones. Análisis empírico

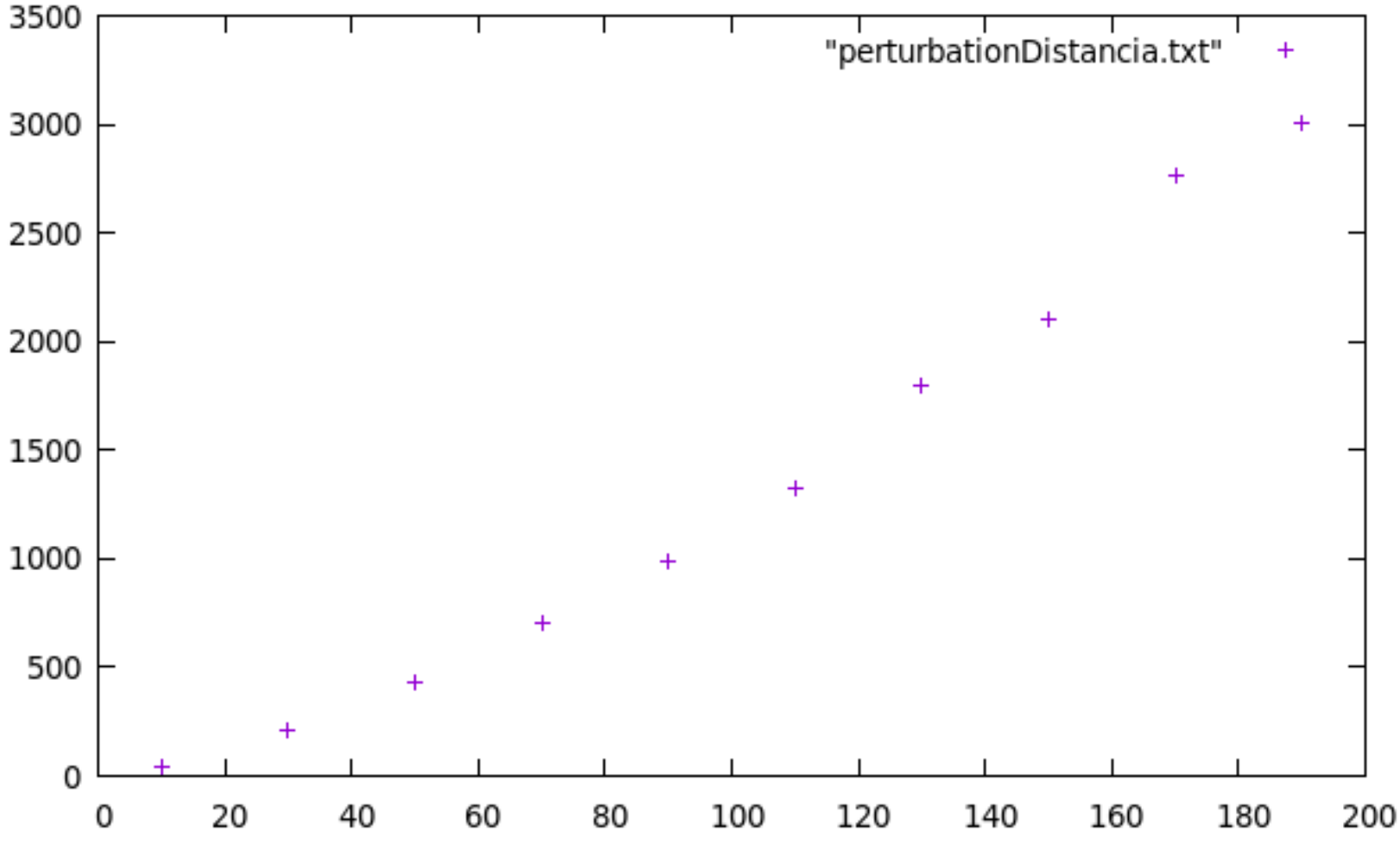
- **Tamaños de prueba:** desde 10 hasta 200 ciudades, con incremento de 20.
- Cada iteración la repetimos 100 veces y hacemos la media, con el fin de eliminar peores y mejores casos.

Enfoque 3: por perturbaciones. Análisis empírico



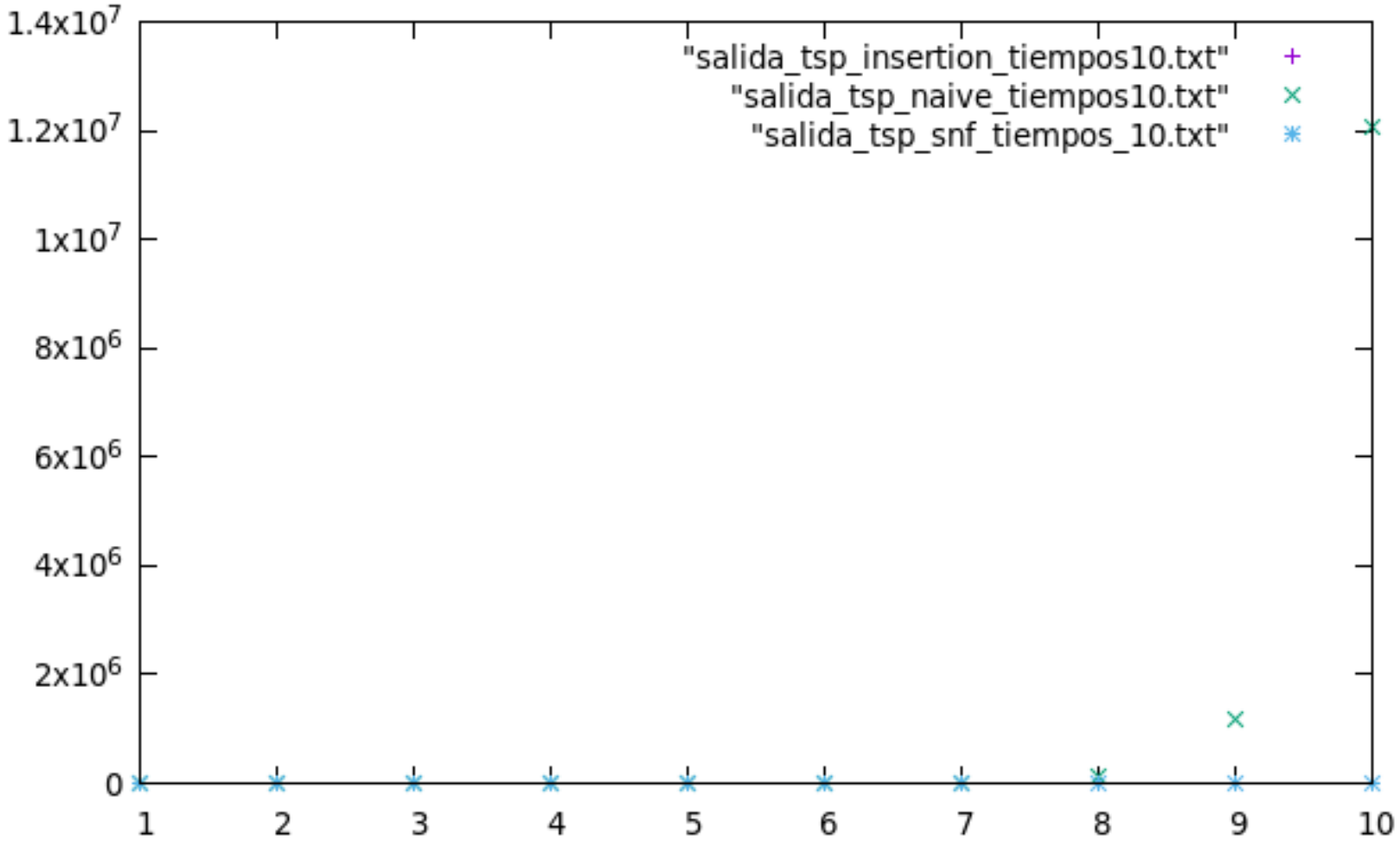
Datos empíricos para viajante de comercio versión perturbaciones, tiempo

Enfoque 3: por perturbaciones. Análisis empírico



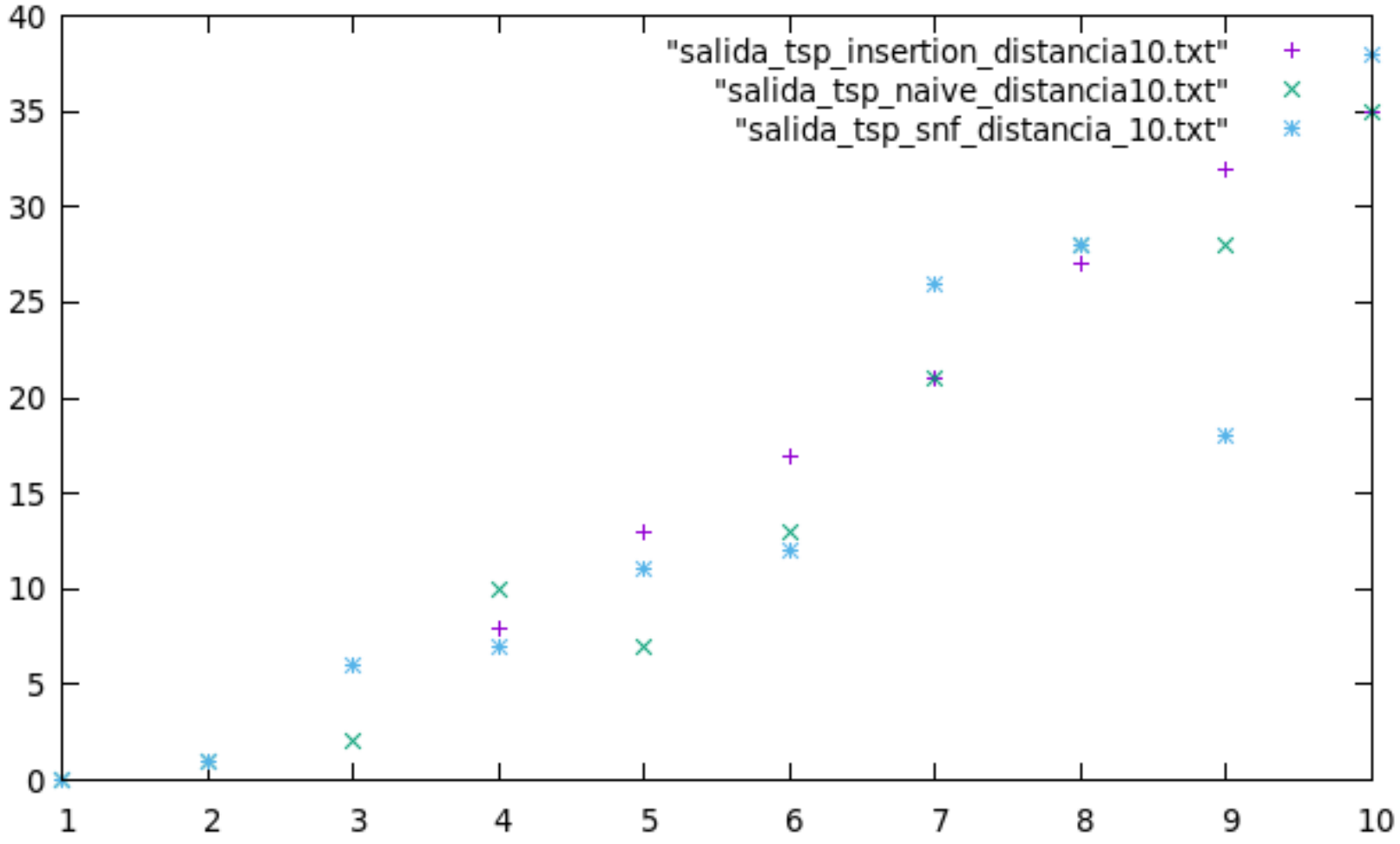
Datos empíricos para viajante de comercio versión perturbaciones, distancia

Comparación de enfoques



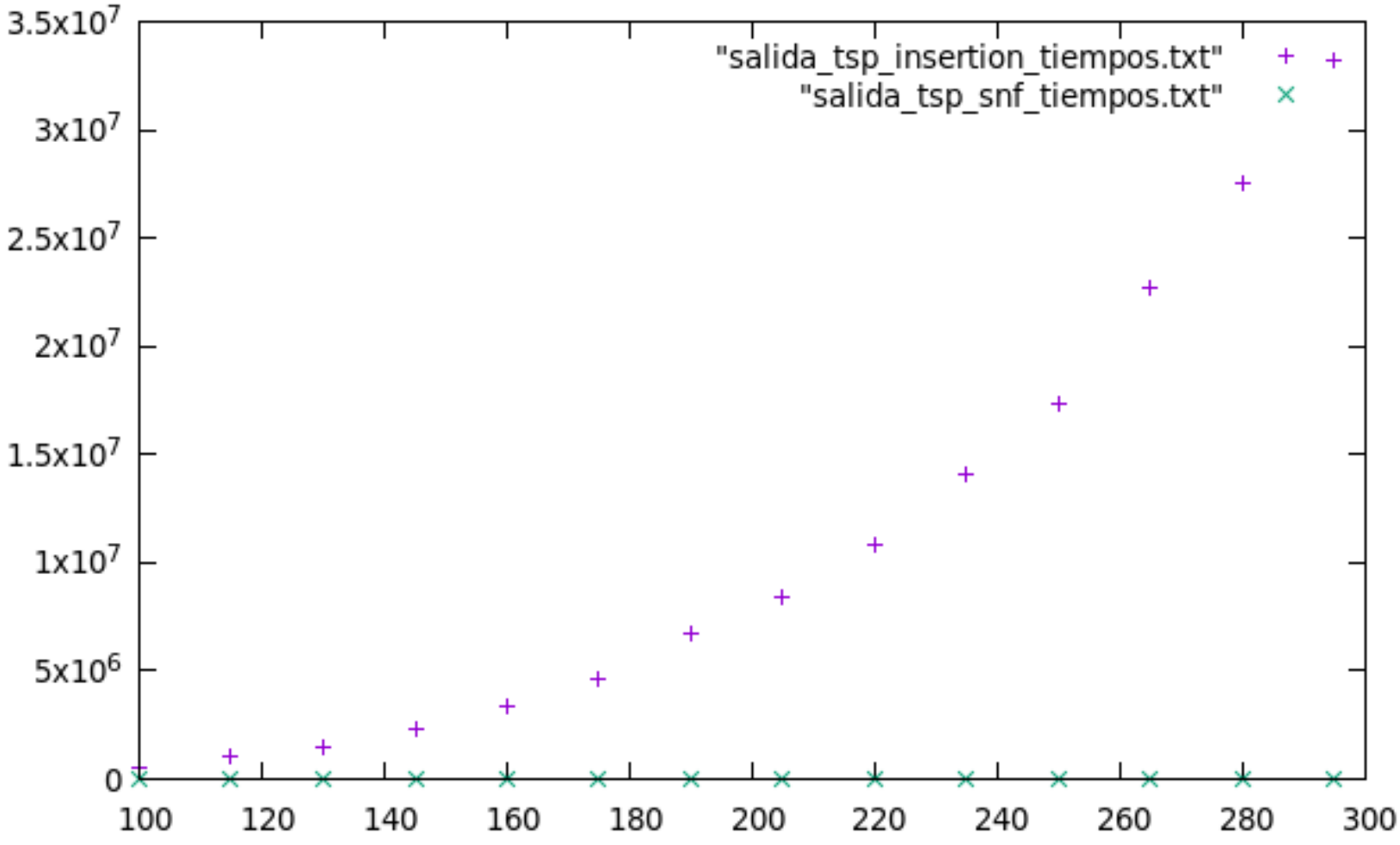
Contraste de datos empíricos: tiempo

Comparación de enfoques



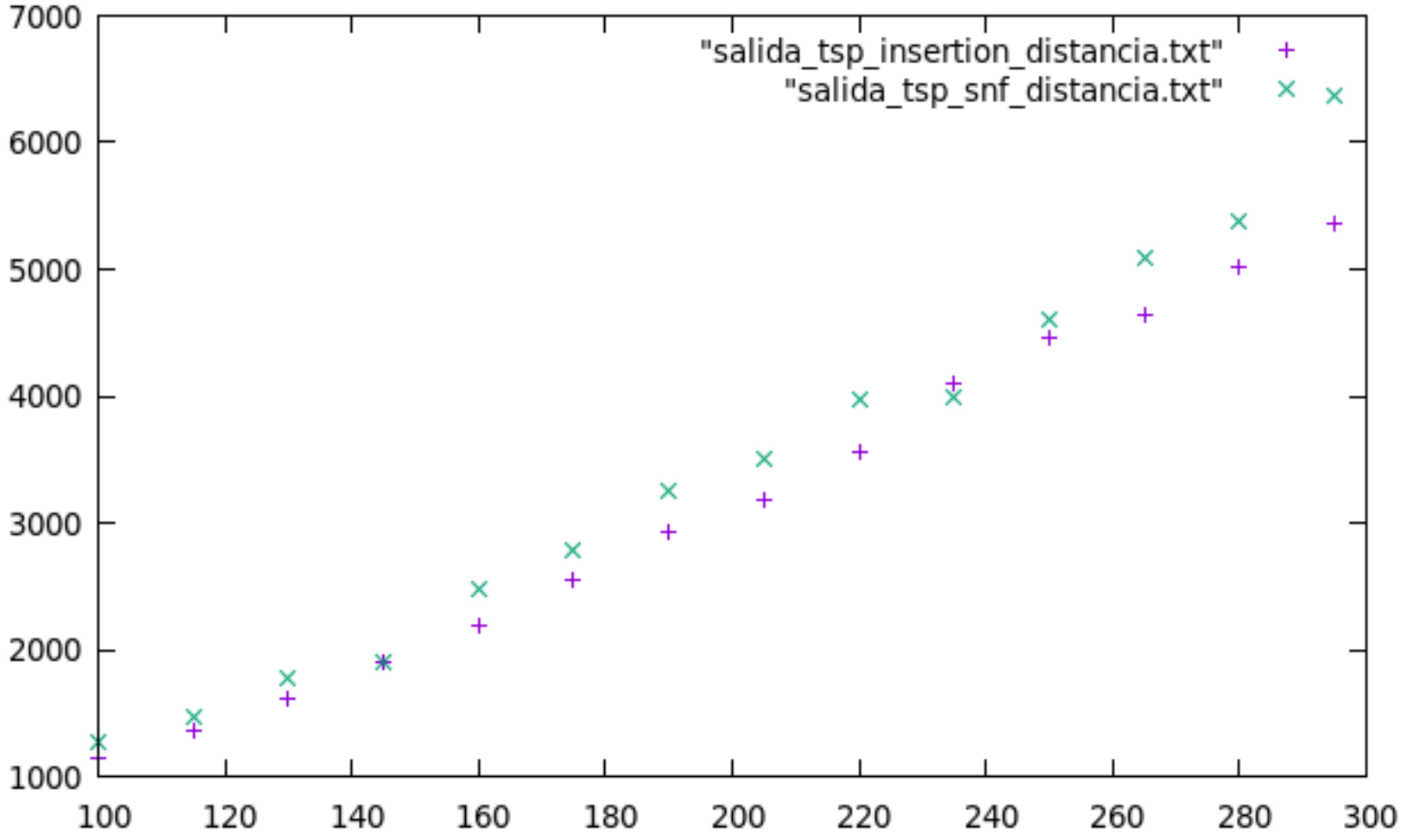
Contraste de datos empíricos: distancia

Comparación de enfoques



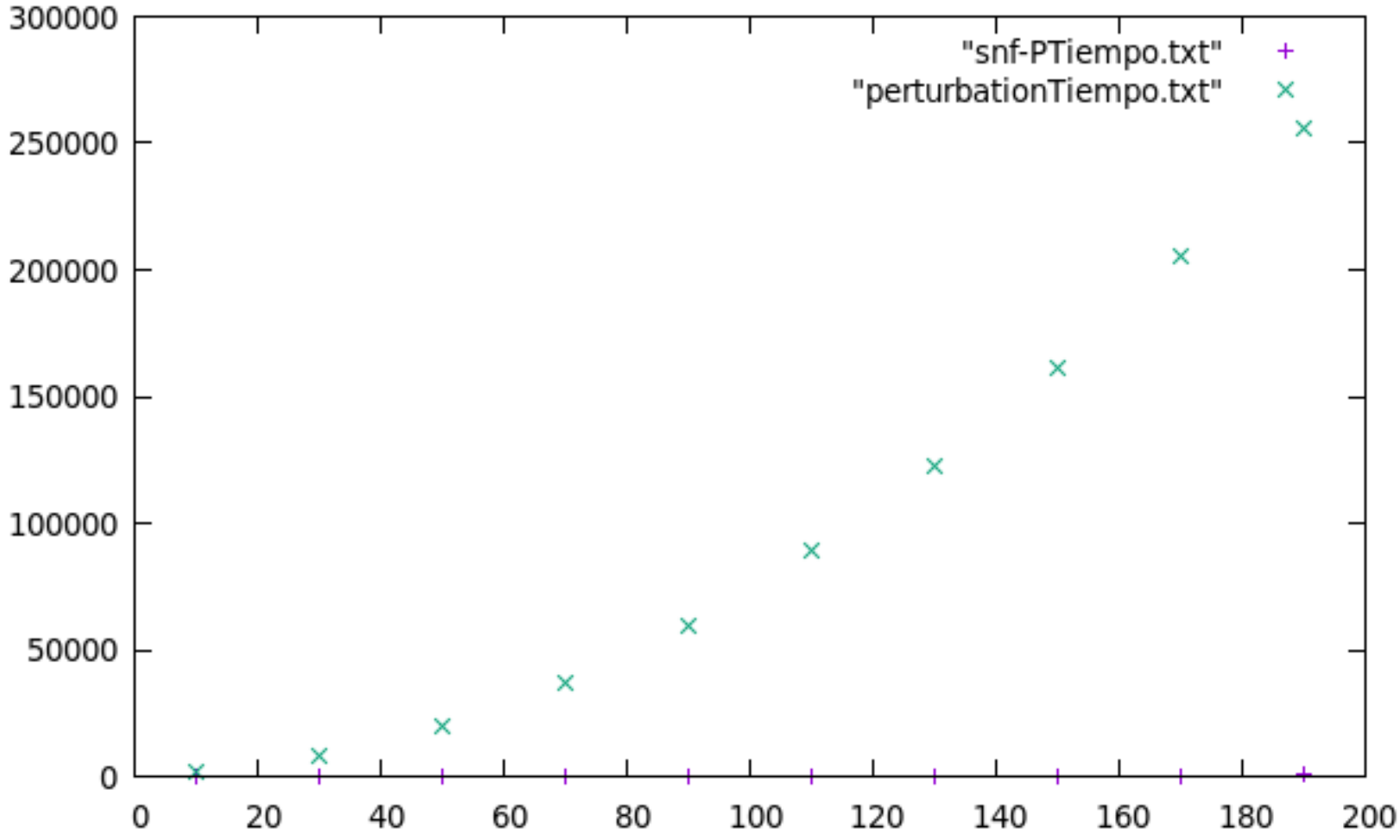
Contraste de datos empíricos (cercanía e inserción): tiempo

Comparación de enfoques



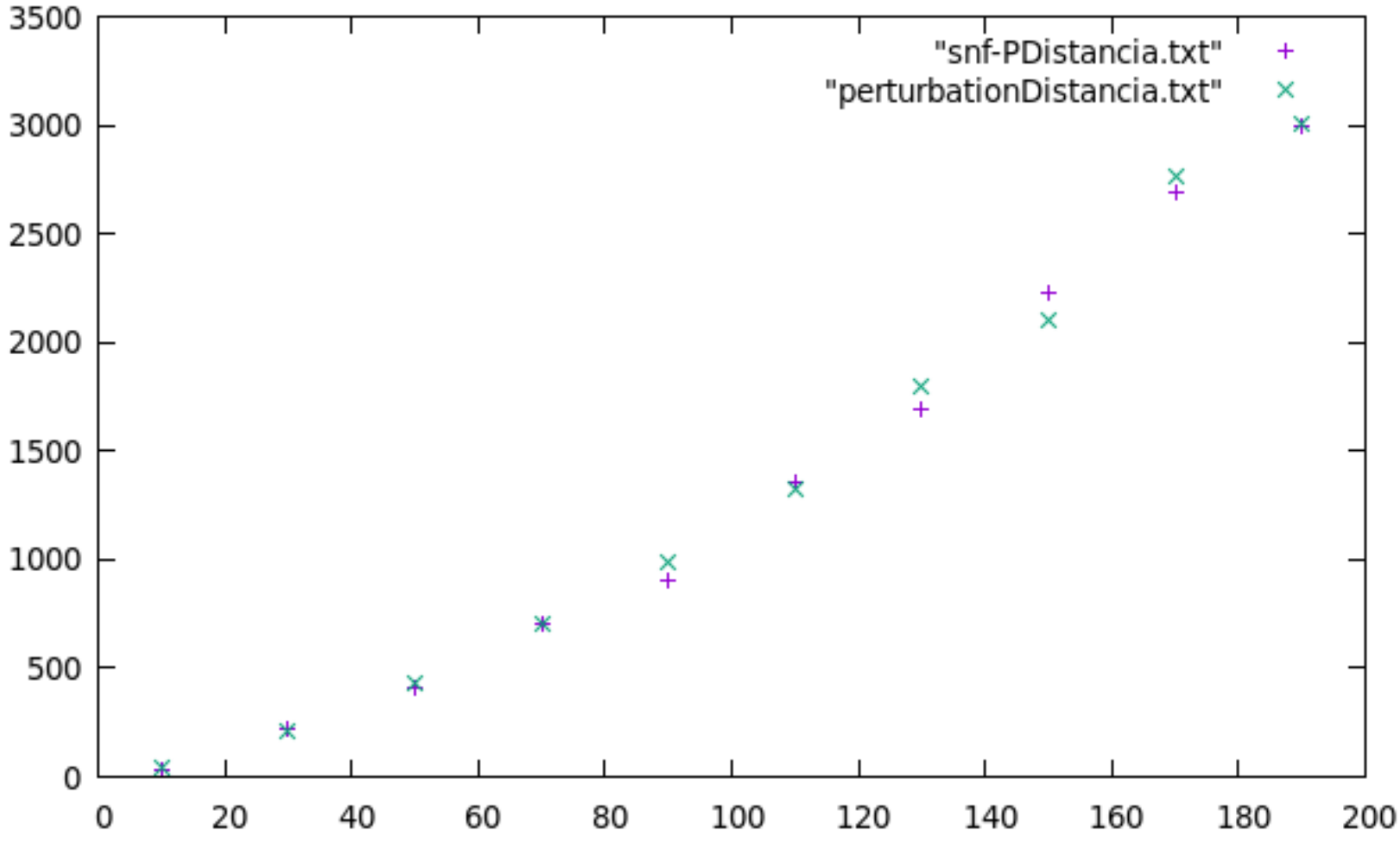
Contraste de datos empíricos (cercanía e inserción): distancia

Comparación de enfoques



Contraste de datos empíricos (cercanía y perturbaciones): tiempo

Comparación de enfoques



Contraste de datos empíricos (cercanía y perturbaciones): distancia

Asignación de tareas (*worker*)

Problema asignado

Problema de la asignación de tareas

Supongamos que disponemos de n trabajadores y n tareas. Sea $c_{ij} > 0$ el coste de asignarle la tarea j al trabajador i. Una asignación válida es aquella en la que a cada trabajador le corresponde una tarea y cada tarea la realiza un trabajador diferente. Dada una asignación válida, definimos el coste de dicha asignación como la suma total de los costes individuales. Diseñe un algoritmo voraz para obtener una asignación de tareas a trabajadores óptima.

Notación

Usaremos la siguiente notación:

- n es el **número de trabajadores**, que coincide con el **número de tareas**.
- C es la **matriz de costes**, de tamaño $n \times n$, en el que el elemento c_{ij} de la matriz es el coste de asignar al trabajador i la tarea j.
- a es el **vector de asignaciones**, de tamaño n. El trabajo asignado al trabajador i estará contenido en el elemento a_i del vector.
- W_a es el coste de una cierta asignación a. Se calcula mediante:

$$W_a = \sum_{i=0}^{n-1} c_{i,a_i}$$

Enfoque 1: inserción

(1)

$$C = \begin{pmatrix} \mathbf{2} & 4 & 3 \\ 1 & 4 & 2 \\ 2 & 7 & 5 \end{pmatrix} \leftarrow$$

$$a = \{0, \ , \ \}$$

$$d = \{\mathrm{T}, \mathrm{T}, \mathrm{T}\}$$

Enfoque 1: inserción

(2)

$$C = \begin{pmatrix} 2 & 4 & 3 \\ 1 & 4 & 2 \\ 2 & 7 & 5 \end{pmatrix} \leftarrow$$

$$a = \{0, \ , \ \}$$

$$d = \{F, T, T\}$$

Enfoque 1: inserción

(3)

$$C = \begin{pmatrix} 2 & 4 & 3 \\ 1 & 4 & \mathbf{2} \\ 2 & 7 & 5 \end{pmatrix} \leftarrow \qquad C = \begin{pmatrix} 2 & 4 & 3 \\ 1 & 4 & 2 \\ 2 & 7 & 5 \end{pmatrix} \leftarrow$$

$$a = \{0, 2, \ \} \qquad a = \{0, 2, \ \}$$

$$d = \{F, T, T\} \qquad d = \{F, T, F\}$$

Enfoque 1: inserción

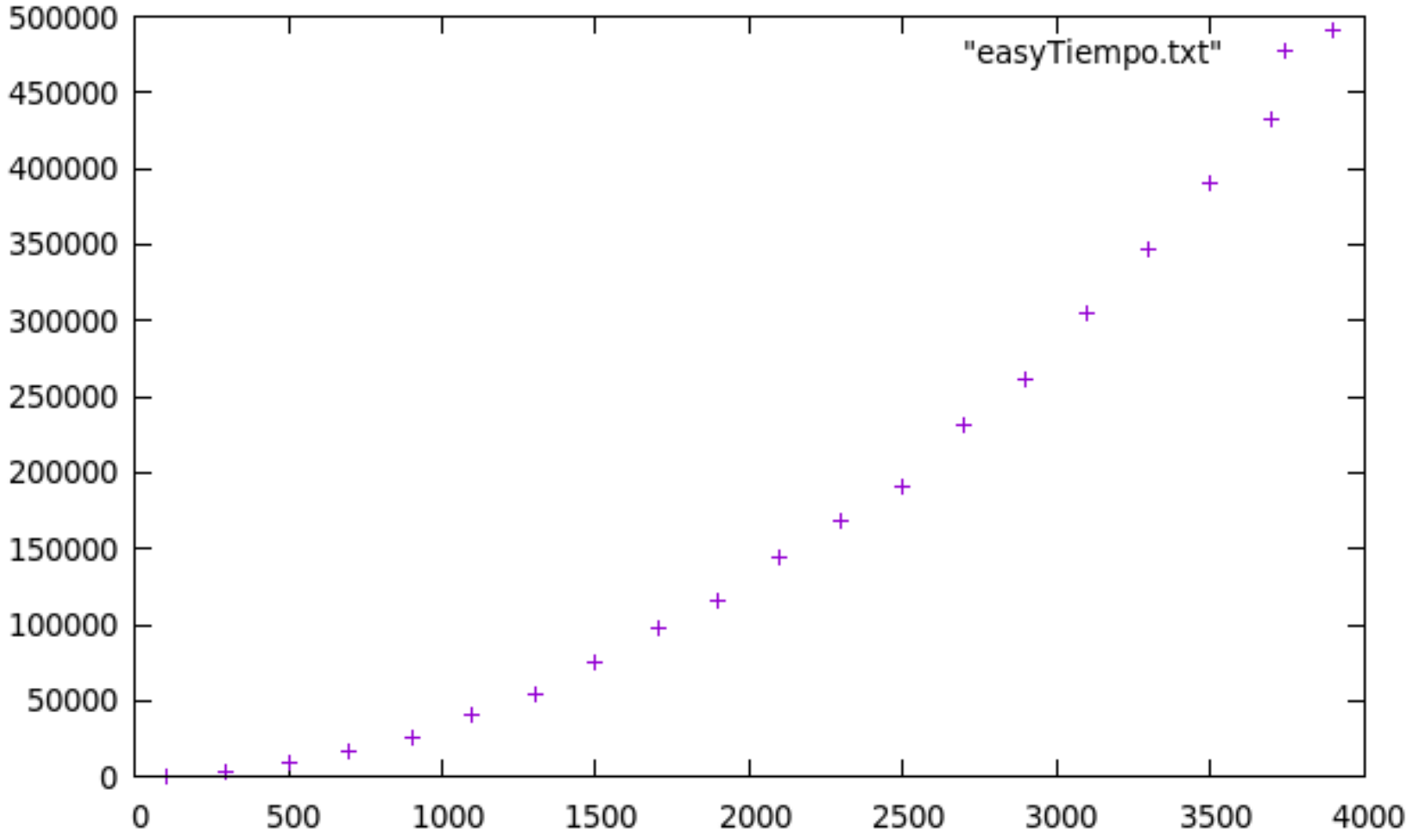
Este enfoque se traduce en un código que comprende las siguientes funciones:

- `get_best_solution`, una función para calcular la asignación especificada anteriormente.
- `find_best_task`, una función auxiliar para calcular la mejor tarea que se encuentre disponible (haciendo uso de un vector de disponibles, `available`), de un conjunto de tareas

Enfoque 1: inserción. Análisis empírico

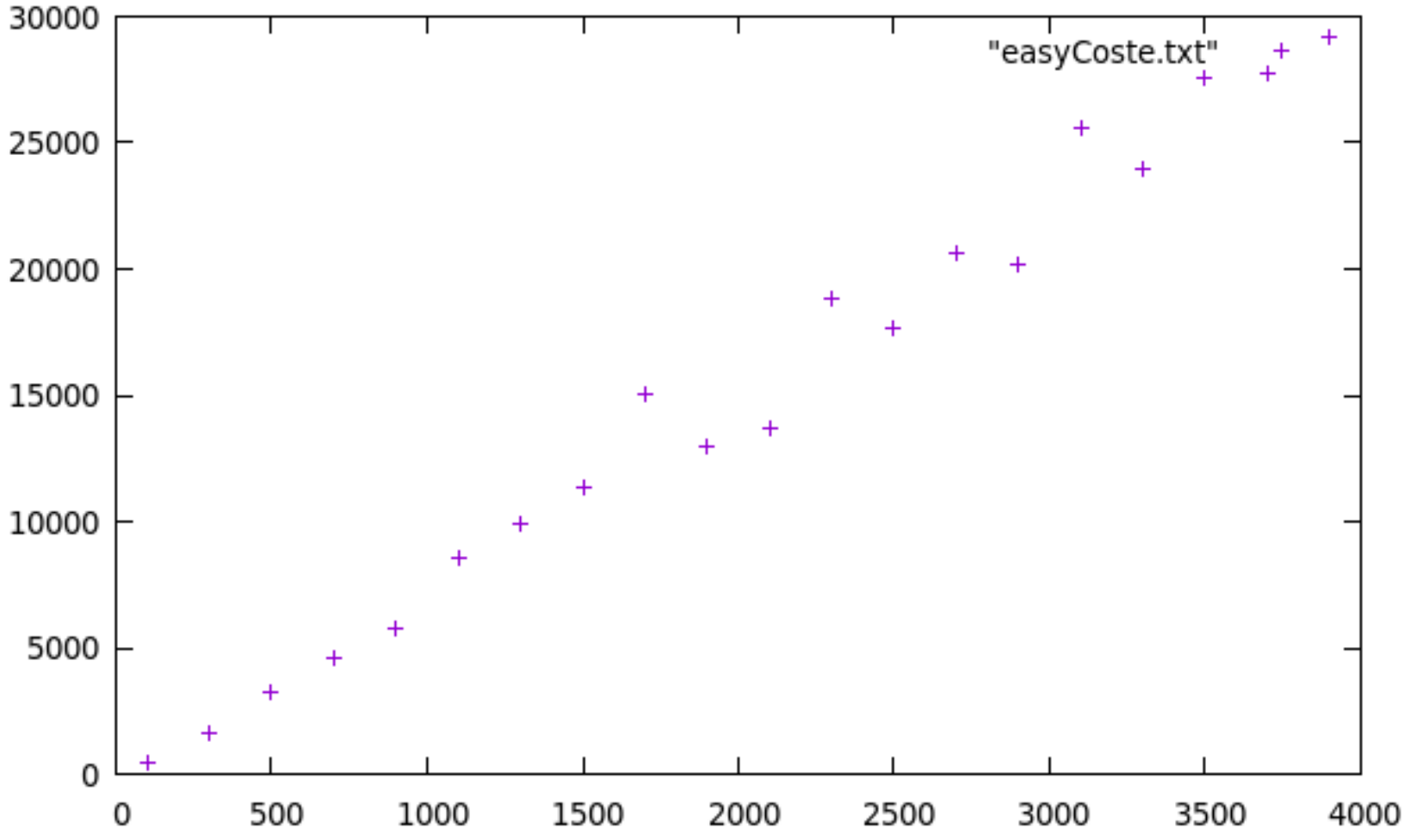
- **Tamaños de prueba:** desde 100 hasta 400 trabajadores, con incremento de 200.
- Cada iteración la repetimos 100 veces y hacemos la media, con el fin de eliminar peores y mejores casos.

Enfoque 1: inserción. Análisis empírico



Datos empíricos para asignación de tareas versión inserción, tiempo

Enfoque 1: inserción. Análisis empírico



Datos empíricos para asignación de tareas versión inserción, coste

Enfoque 2: permutaciones

Este algoritmo consiste en, partiendo de una asignación, ir modificándola hasta encontrar una mejor, es decir, con una **ganancia** positiva.

Enfoque 2: permutaciones

(1)

Buscamos elemento con mayor coste de todas las asignaciones: a_m, donde m es la posición del elemento en el vector a de asignaciones.

Enfoque 2: permutaciones

(2)

Vamos permutando con a_m:

$$a'_0 = \{a_m, a_1, ..., a_{m-1}, a_0, a_{m+1}, ..., a_{n-1}\},$$

$$a'_1 = \{a_0, a_m, a_2, ..., a_{m-1}, a_1, a_{m+1}, a_{n-1}\}, ...$$

Tomamos la que tenga mayor **ganancia** (p con $W_{a'_p}$ menor).

Enfoque 2: permutaciones

(3)

Repetimos número **arbitrario** de veces: cada iteración, obtenemos solución mejor, o misma solución.

Enfoque 2: permutaciones

Podemos partir de la asignación que proporciona el enfoque por *inserción*.

Enfoque 2: permutaciones

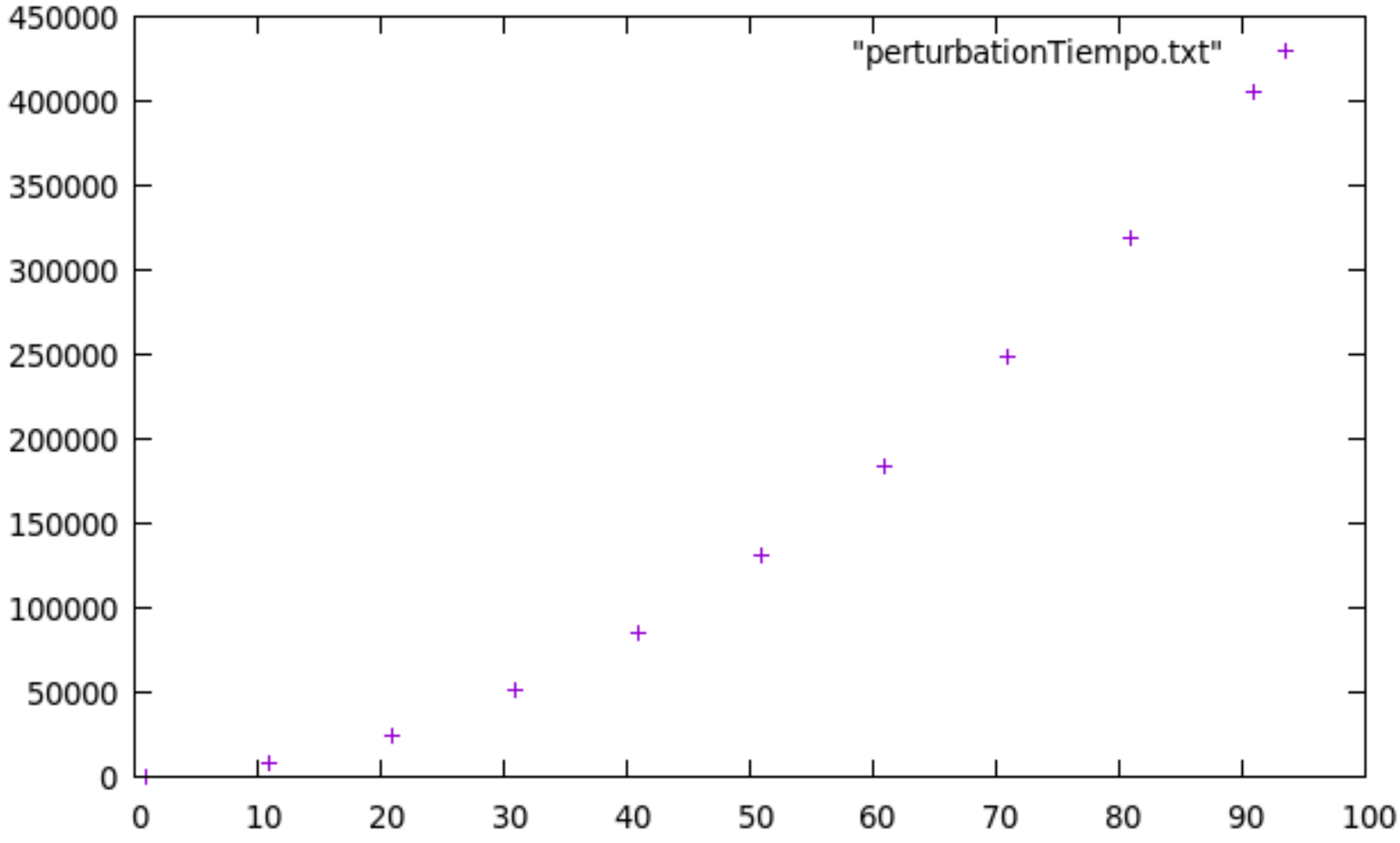
Este enfoque se traduce en un código que comprende las siguientes funciones:

- `get_best_solution`, de nuevo nuestra función principal para resolver el problema.
- `find_worst_worker`, una función auxiliar para encontrar el trabajador que tiene la *peor tarea asignada*, es decir, el que tiene mayor costo (1).
- `find_best_permutation`, una función auxiliar que, de entre todas las permutaciones posibles, encuentra la que tiene una ganancia mayor (2).
- `permutate`, una función auxiliar que efectúa la permutación, es decir, modifica el vector a de asignaciones para efectuar el intercambio encontrado.

Enfoque 2: permutaciones. Análisis empírico

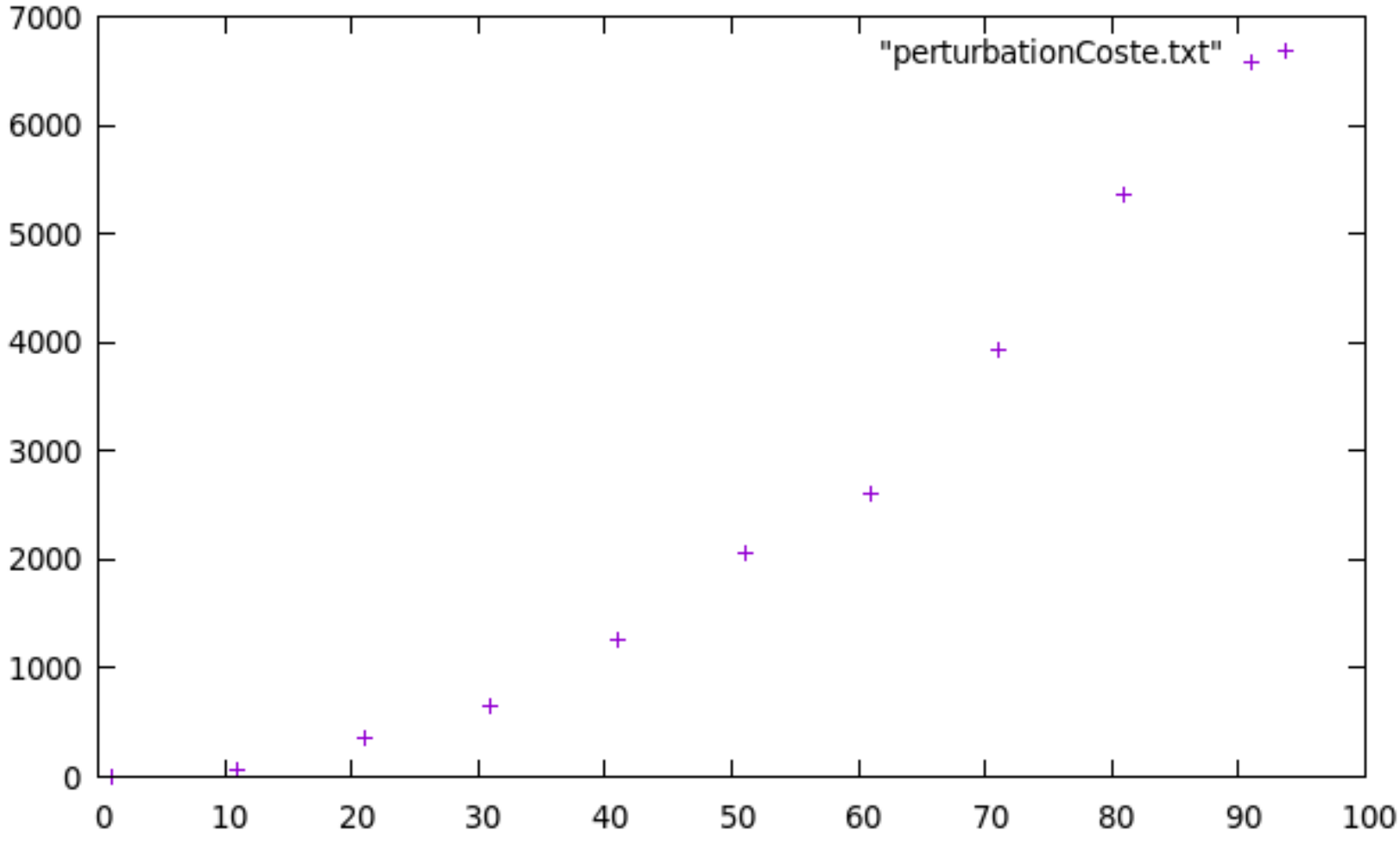
- **Tamaños de prueba:** desde 1 hasta 100 trabajadores, con incremento de 10.
- Cada iteración la repetimos 100 veces y hacemos la media, con el fin de eliminar peores y mejores casos.

Enfoque 2: permutaciones. Análisis empírico



Datos empíricos para asignación de tareas versión permutaciones, tiempo

Enfoque 2: permutaciones. Análisis empírico



Datos empíricos para asignación de tareas versión permutaciones, coste

Comparación de enfoques

Ambos enfoques son *greedy*, e intentan acercarse a la mejor solución.

$$C = \begin{pmatrix} 1 & 2 \\ 2 & 10 \end{pmatrix}$$

Coste asignación por inserción: 11
Coste óptimo: 4

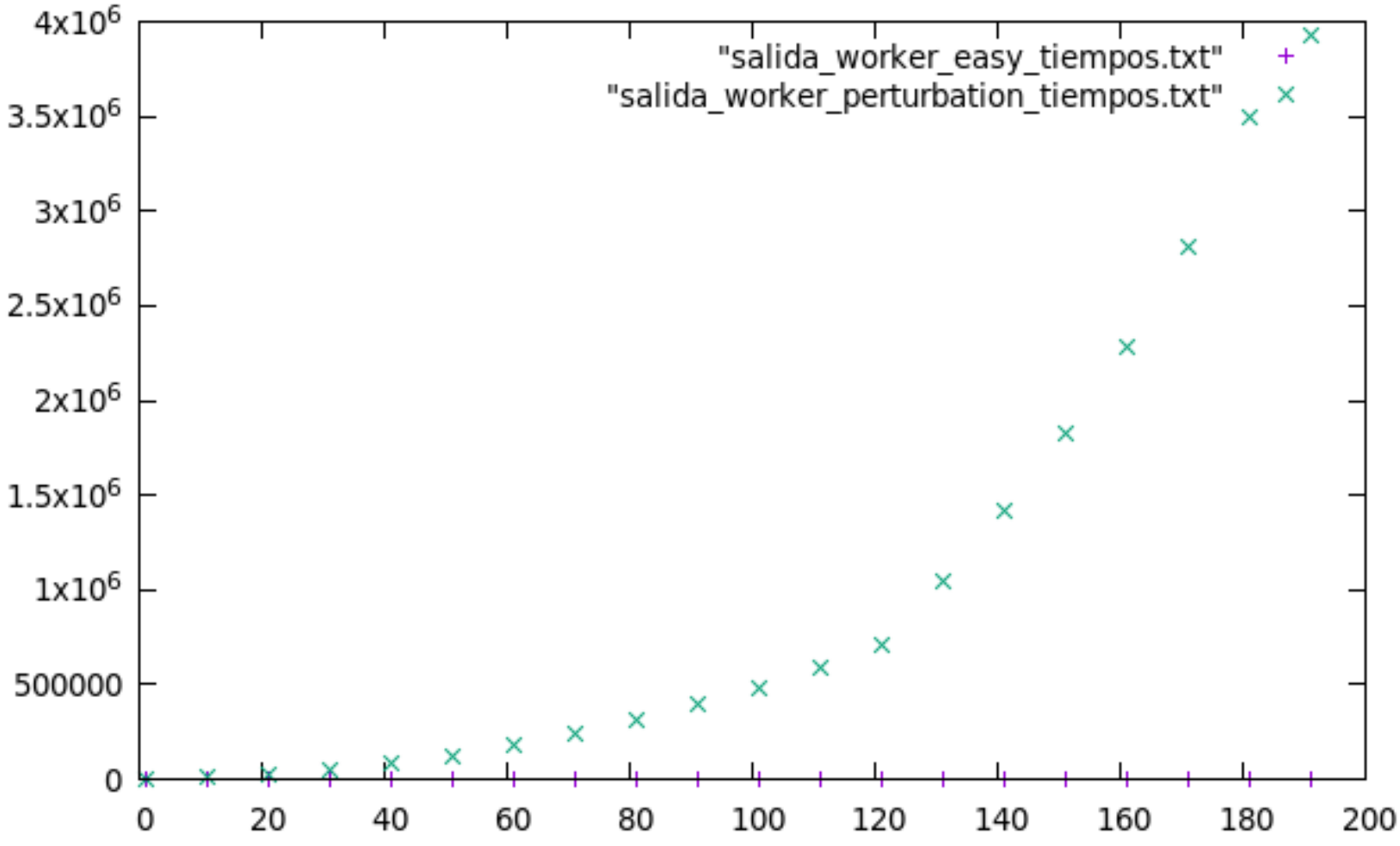
Enfoque con permutaciones nos conduciría a solución óptima en una iteración.

Comparación de enfoques

Enfoque con permutaciones es especialmente interesante cuando partimos de una cantidad muy grande de datos, ya que tener una estimación inicial puede ser complicado.

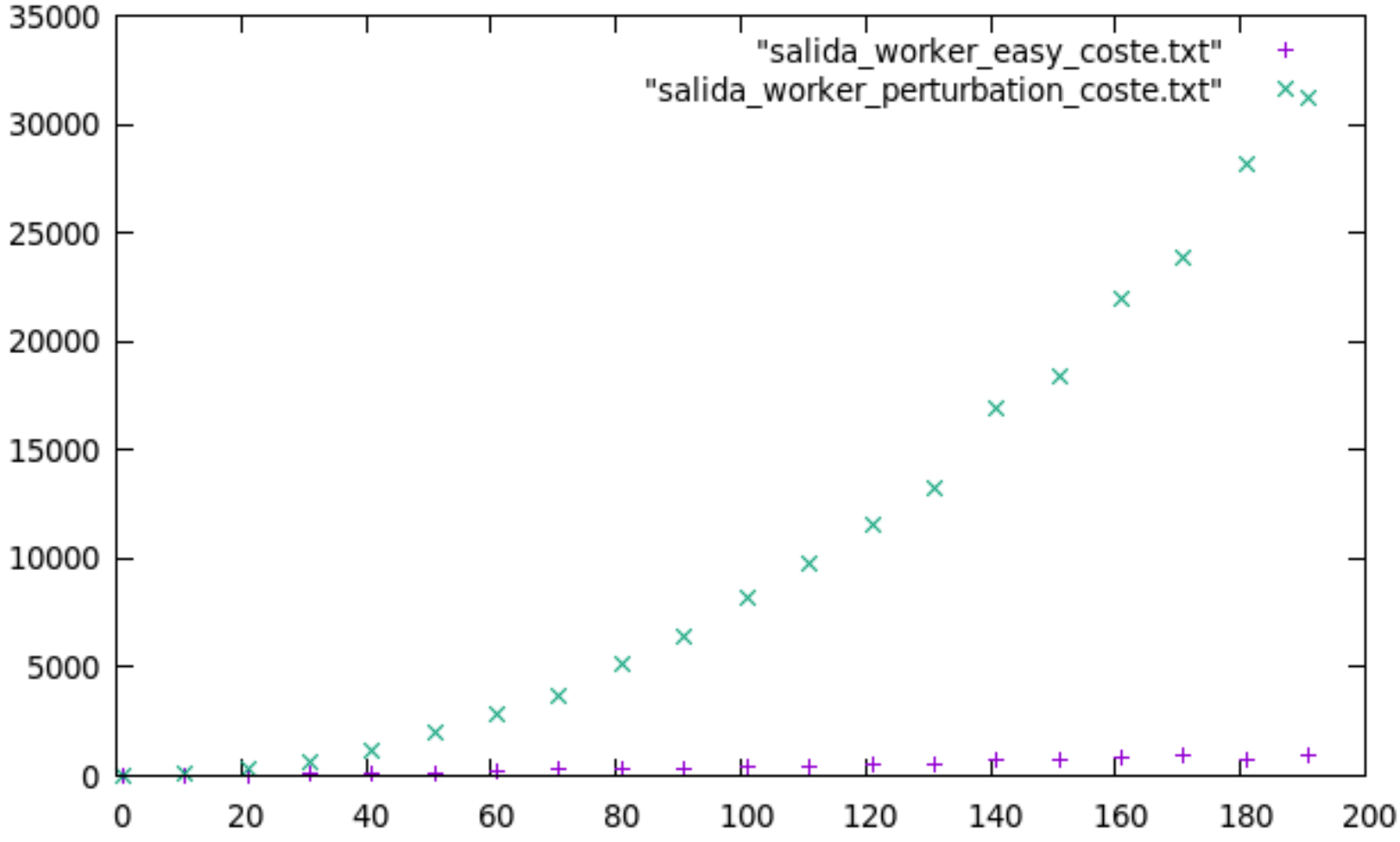
Podemos partir de asignación cualquiera y mejorarla con sucesivas iteraciones.

Comparación de enfoques



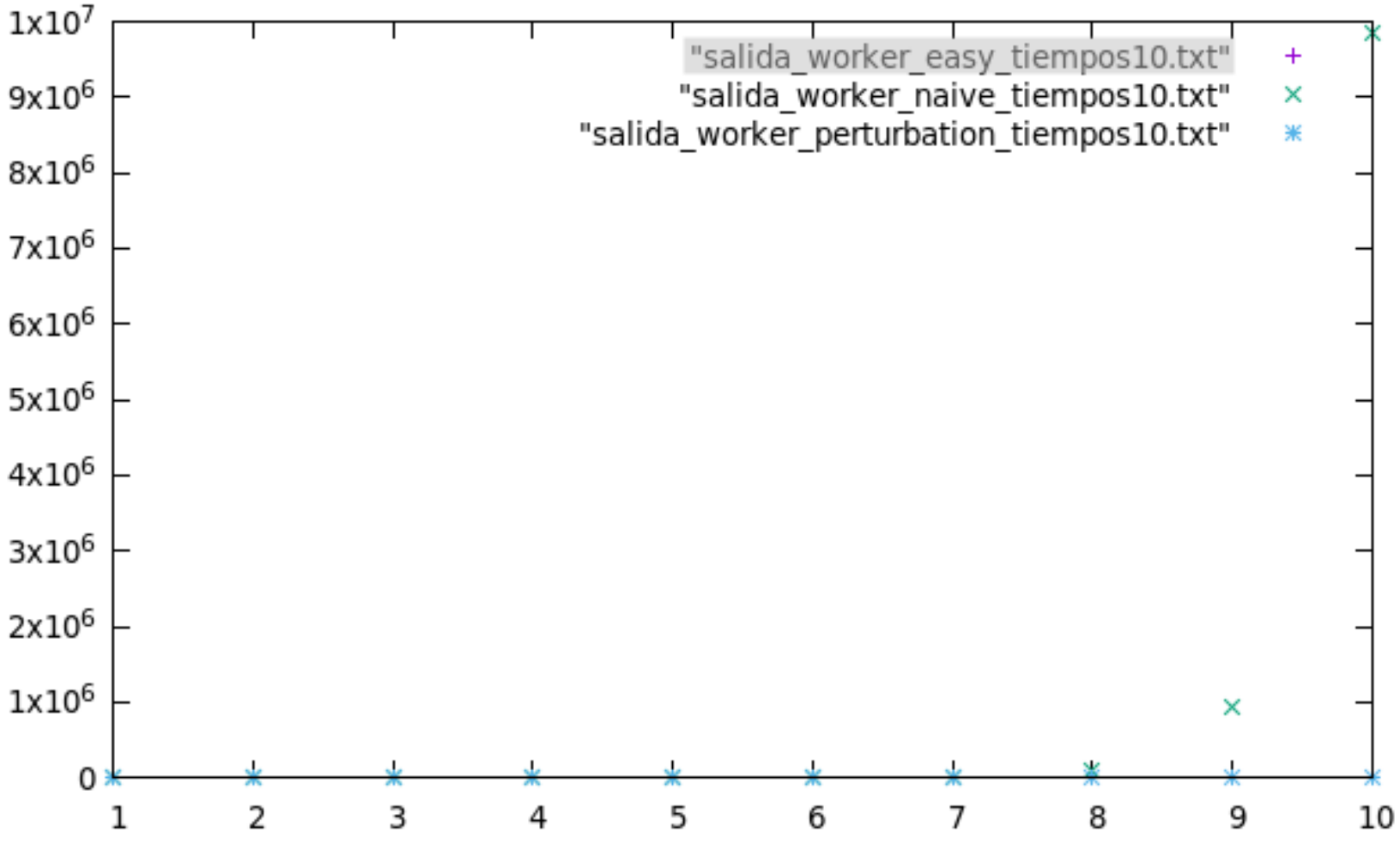
Contraste de datos empíricos (inserción y permutaciones): tiempo

Comparación de enfoques



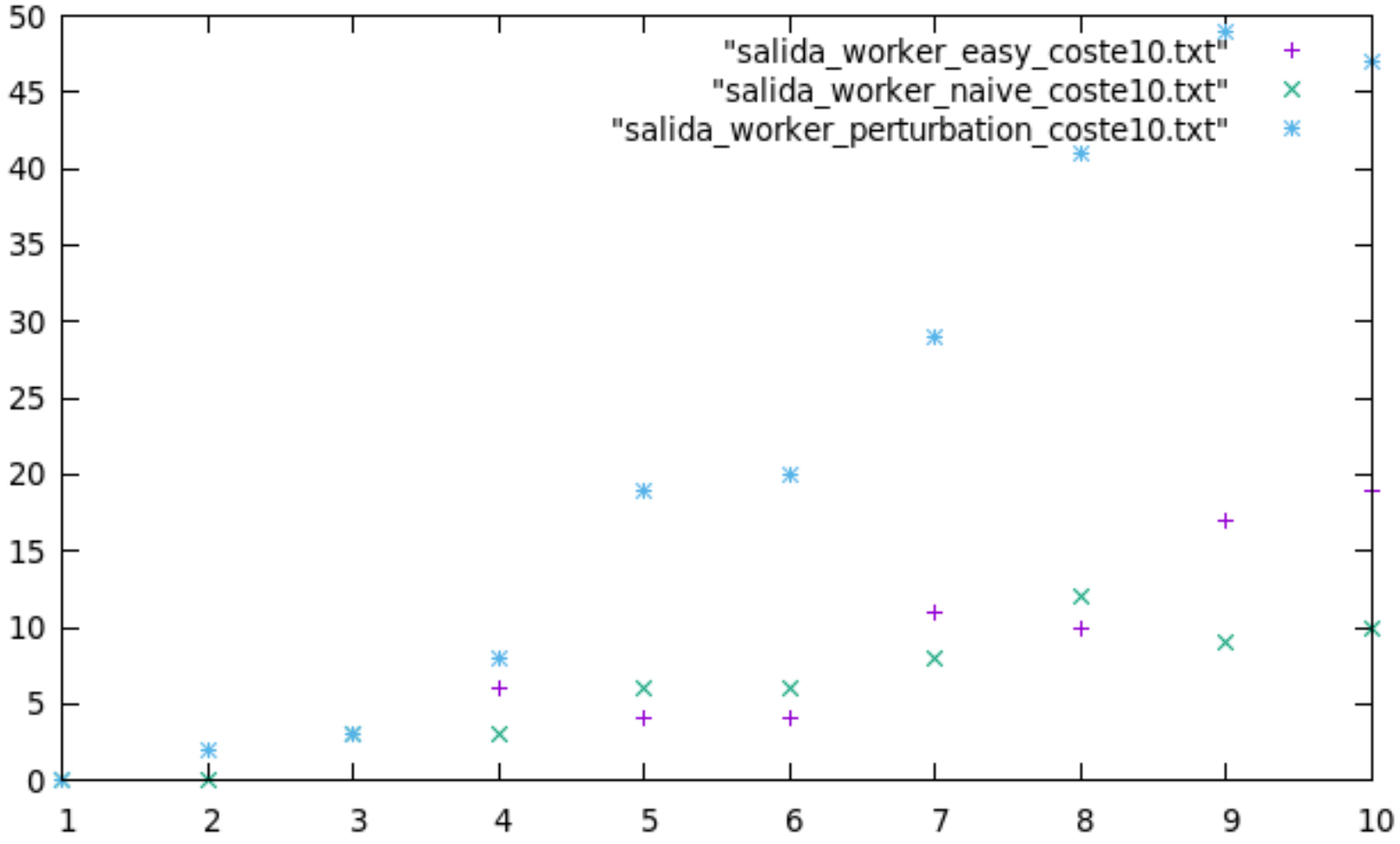
Contraste de datos empíricos (inserción y permutaciones): coste

Comparación de enfoques



Contraste de datos empíricos (inserción, permutaciones y fuerza bruta): tiempo

Comparación de enfoques



Contraste de datos empíricos (inserción, permutaciones y fuerza bruta): coste

Conclusiones

Conclusiones

Hemos aprendido a crear algoritmos voraces para resolver problemas que, en su versión en fuerza bruta, tienen una complejidad muy elevada, con diversos **enfoques *greedy***.

Hay problemas en el que encontrar la solución óptima es muy costoso. Podemos entonces “acercarnos” a ella mediante los **algoritmos *greedy***, y especialmente haciendo uso de enfoques con **permutaciones** a partir de soluciones iniciales.

Muy relevante al trabajar con grandes cantidades de datos.