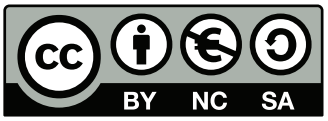
Práctica 3

Algoritmos voraces (*greedy*)

Algorítmica

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

UNIVERSIDAD
DE GRANADA



Práctica 3

Algoritmos voraces (*greedy*)

Algorítmica

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

UNIVERSIDAD
DE GRANADA

Índice

I Introducción

Esta **práctica 3**, de desarrollo de algoritmos *greedy*, consiste en dos partes principales:

- **Problema común:** problema del viajante de comercio.
- **Problema asignado:** asignación de tareas.

Objetivo de esta práctica

En esta práctica, pretenderemos apreciar la utilidad de los algoritmos voraces (*greedy*) para la resolución de problemas de forma eficiente, en algunos casos obteniendo soluciones óptimas y en otros aproximaciones.

Para ello, daremos diversas soluciones *greedy* a los problemas asignados, y compararemos la bondad en la solución y la eficiencia de estos algoritmos, dando especial importancia al uso de **permutaciones** para acercarnos a soluciones cada vez mejores con sucesivas iteraciones, como veremos en los últimos enfoques de cada problema (*sección II, apartados 1.3* y *2.2*).

II Desarrollo

A continuación, estudiaremos los dos algoritmos propuestos.

1 Problema del viajante de comercio (TSP)

> Dado un conjunto de ciudades y una matriz con las distancias entre ellas, buscar el recorrido mínimo que que pase por todas ellas una vez y vuelva al punto de partida.
>
> Formalmente: dado un grafo *G*, conexo y ponderado, se trata de hallar el *ciclo hamiltoniano* de mínimo peso de ese grafo.

Elementos comunes

A lo largo de la solución del problema usaremos la siguiente notación:

- n es el **número de ciudades**.
- D es la **matriz de distancias**.
- r es el **vector de recorrido**, que contiene un itinerario que pasa por todas las ciudades, es decir, n elementos no repetidos.
- W_r es el **coste** de un recorrido, es decir, la distancia de un recorrido r.

1.1 Enfoque por cercanía

Este algoritmo *greedy* es muy simple:

1. Partimos de un nodo cualquiera.
2. Encontramos el nodo más cercano a este nodo, y lo añadimos al recorrido.
3. Repetimos el proceso hasta cubir todos los nodos (encontrando el siguiente nodo más cercano).
4. Añadimos el nodo de vuelta.

Hacemos uso del siguiente código:

```
vector<common::Point> get_best_solution(vector<common::Point> points){
    vector<common::Point> road;                 // Solucion que vamos a construir
    vector<common::Point> points_left = points; // Puntos que quedan por insertar a la
    solucion
```

```
    // Parto siempre del primer punto del vector
    road.push_back(points_left[0]);
    points_left.erase(points_left.begin() + 0);
    // Voy construyendo la solucion, sacando puntos de points_left y colocandolos en road
    while(points_left.size() > 0){
        double min_distance = common::distance(road[road.size() -1], points_left[0]);
        int min_pos = 0;
        // Buscamos el punto mas cercano
        for(int i = 0; i < points_left.size(); i++){
            double current_distance = common::distance(road[road.size() -1], points_left[i]);
            if(current_distance < min_distance){
                min_distance = current_distance;
                min_pos = i;
        }
    }
    // Insertamos el punto mas cercano a la solucion y lo quitamos de los puntos que faltan
    road.push_back(points_left[min_pos]);
    points_left.erase(points_left.begin() + min_pos);
    }
    return road;
}
```

En el que utilizamos las siguientes funciones y estructuras de datos:

- 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.

  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`.

El código completo se encuentra en la ***sección IV: Anexo I*** de este documento.

Análisis empírico

Los tamaños de prueba para ejecutar el algoritmo han ido desde 100 hasta 4000 ciudades, cada vez con incremento de 200. A su vez cada iteración la hemos hecho 100 veces y hemos calculado la media, con el fin de eliminar los mejores y peores casos.

Los datos obtenidos, calculando por un lado el tiempo y por otro la distancia han sido estos:

Gráfico 1.1.1. Datos empíricos para viajante de comercio versión cercanía, tiempo

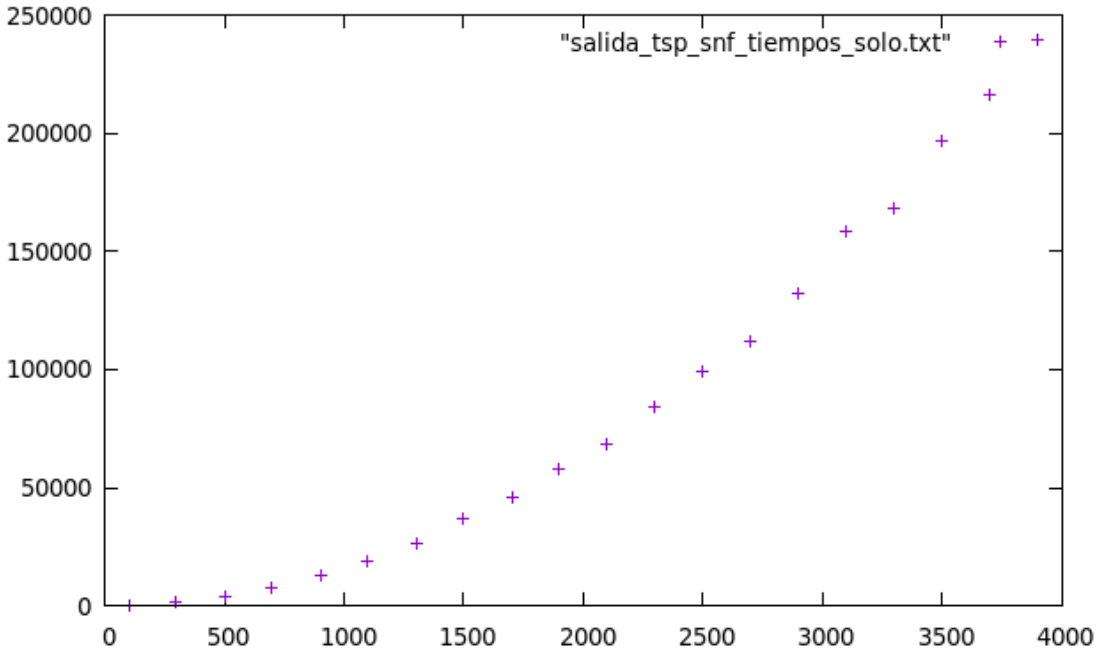
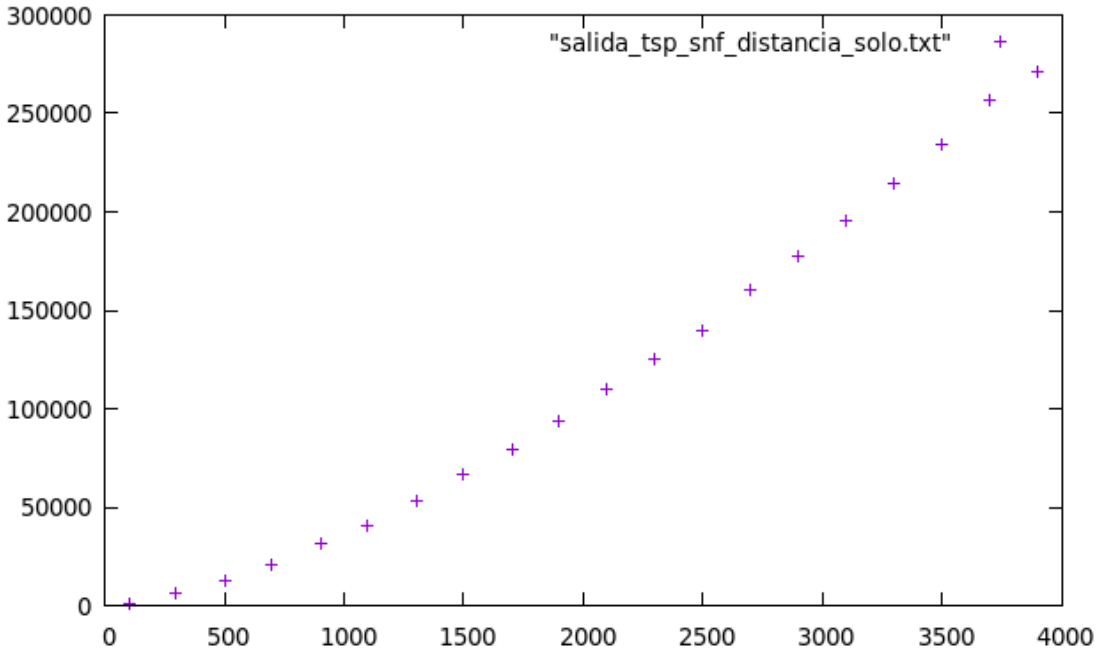


Gráfico 1.1.2. Datos empíricos para viajante de comercio versión cercanía, distancia.



Los datos de las gráficas se encuentran en la ***sección IV: Anexo II*** de este documento.

1.2 Enfoque por inserción

Este algoritmo *greedy* funciona tomando un recorrido dado e insertando nodos de modo que el recorrido sea mínimo.

```
vector<common::Point> get_best_solution(vector<common::Point> points){
    vector<common::Point> candidates = points;  // Vector con los candidatos
    vector<common::Point> road;                 // Vector con la solucion parcial

    // Tomo los tres puntos extremos del plano y los coloco como solucion parcial inicial
    int most_north = get_most_north(candidates);
    int most_east = get_most_east(candidates);
    int most_west = get_most_west(candidates);

    road.push_back(candidates[most_north]);
    road.push_back(candidates[most_west]);
    road.push_back(candidates[most_east]);

    candidates.erase(candidates.begin() + most_north);
    candidates.erase(candidates.begin() + most_west);
    candidates.erase(candidates.begin() + most_east);

    // Construyo las soluciones parciales
    while(candidates.size() > 0){
        // Tomo el mejor candidato para la siguiente iteracion
        vector<int> best_candidate_and_pos = get_best_candidate(road, candidates);
        int best_candidate = best_candidate_and_pos[0];
        int best_pos = best_candidate_and_pos[1];

        // Hago el traspase de candidatos a solucion parcial
        road.insert(road.begin() + best_pos, candidates[best_candidate]);
        candidates.erase(candidates.begin() + best_candidate);
    }

    return road;
}
```

En este código, tenemos que:

- `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.

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`.

Análisis empírico

Los tamaños de prueba para ejecutar el algoritmo han ido desde 3 hasta 200 ciudades, cada vez con incremento de 10. A su vez cada iteración la hemos hecho 100 veces y hemos calculado la media, con el fin de eliminar los mejores y peores casos.

Los datos obtenidos, calculando por un lado el tiempo y por otro la distancia han sido estos:

Gráfico 1.2.1. Datos empíricos para viajante de comercio versión inserción, tiempo

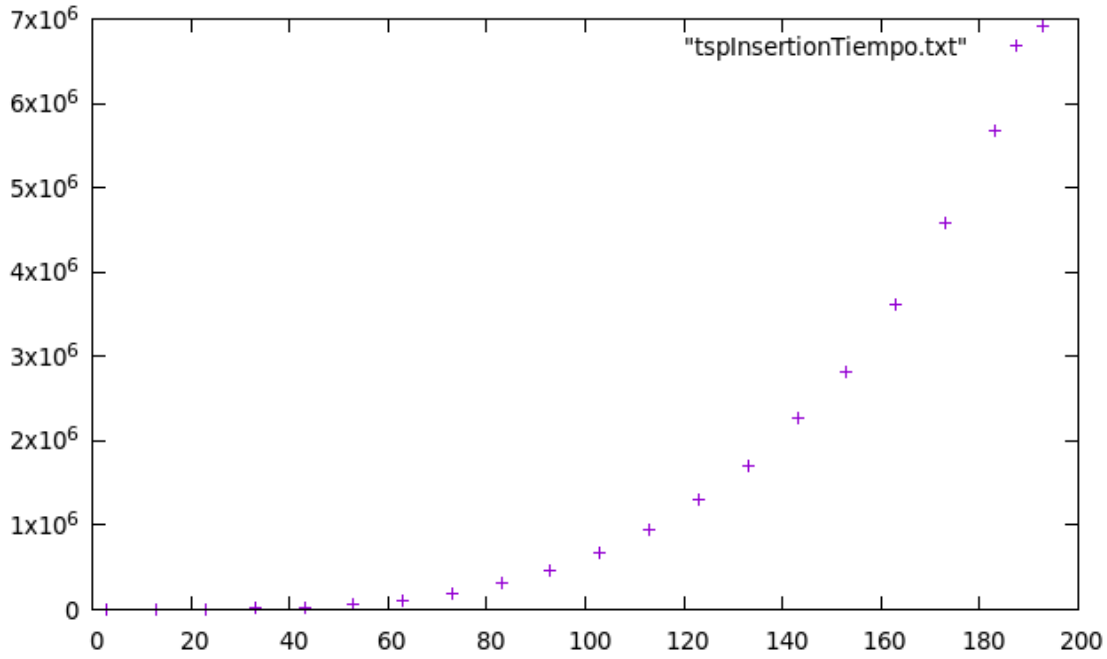
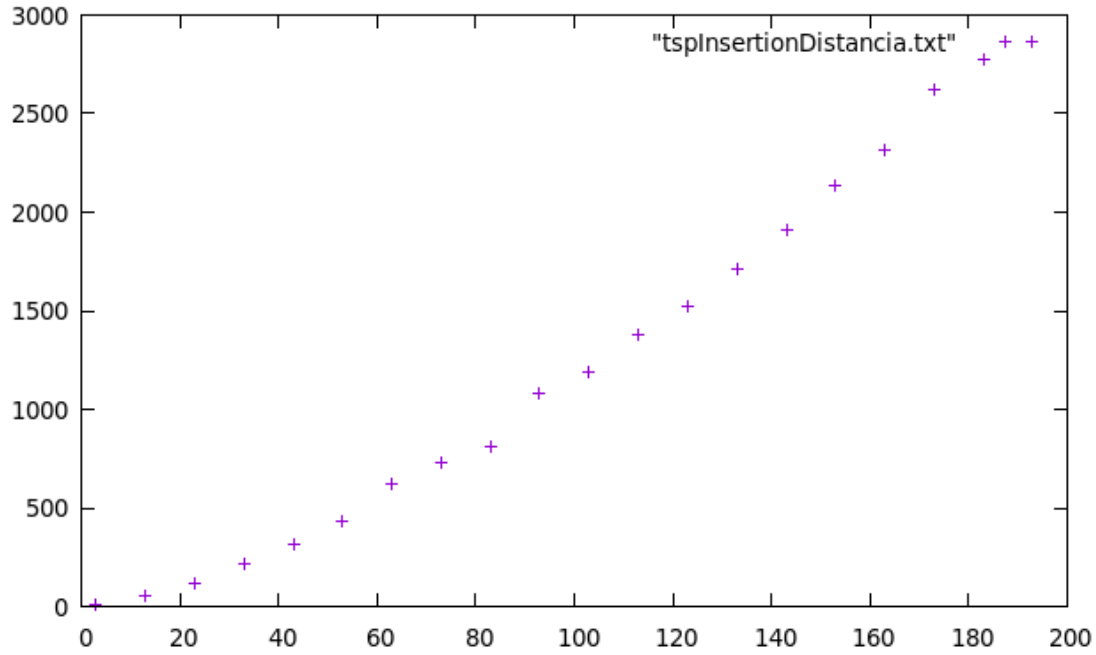


Gráfico 1.2.2. Datos empíricos para viajante de comercio versión inserción, distancia



Los datos de las gráficas se encuentran en la ***sección IV: Anexo II*** de este documento.

1.3 Enfoque por perturbaciones

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

```
vector<common::Point> get_best_solution(vector<common::Point> points, int perturbations){
    // Tomo la solucion dada por el snf
    vector<common::Point> base_road = get_snf_solution(points);

    // Perturbo el numero de veces indicada
    for(int i = 0; i < perturbations; i++){
        // Calculo la posicion respecto a la que perturbar
        int pos = get_worst_node(points);

        // Perturbo el camino base
        perturbate(base_road, pos);
    }

    return base_road;
}
```

En este código, tenemos que:

- `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.

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.

Análisis empírico

Los tamaños de prueba para ejecutar el algoritmo han ido desde 10 hasta 200 ciudades, cada vez con incremento de 20. A su vez cada iteración la hemos hecho 100 veces y hemos calculado la media, con el fin de eliminar los mejores y peores casos.

Los datos obtenidos, calculando por un lado el tiempo y por otro la distancia han sido estos:

Gráfico 1.3.1. Datos empíricos para viajante de comercio versión perturbaciones, tiempo

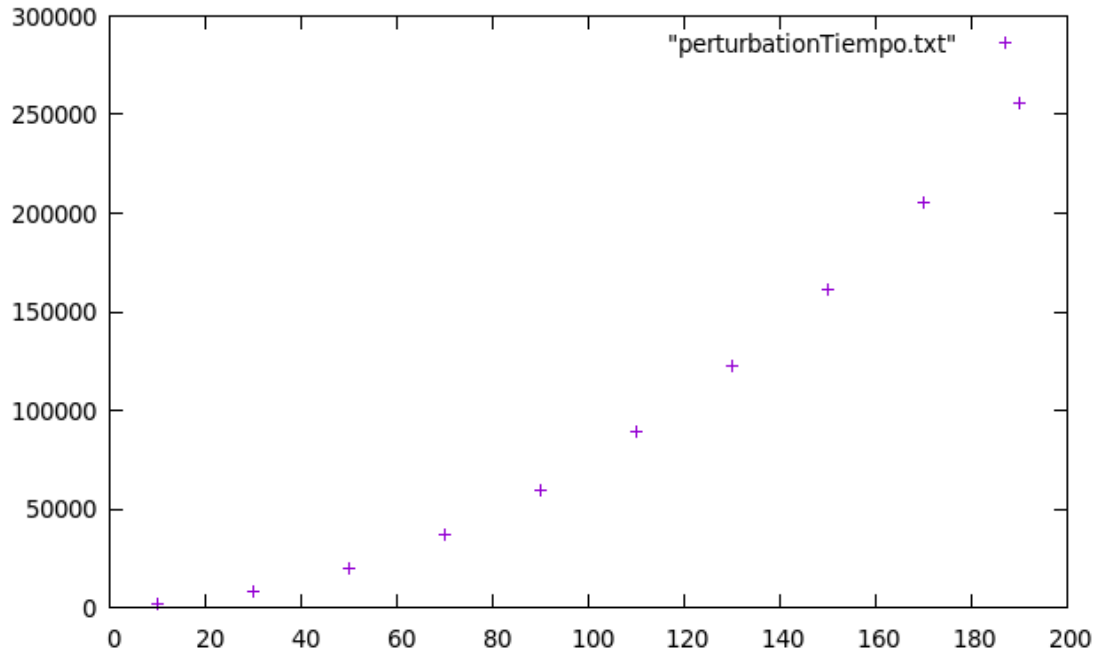
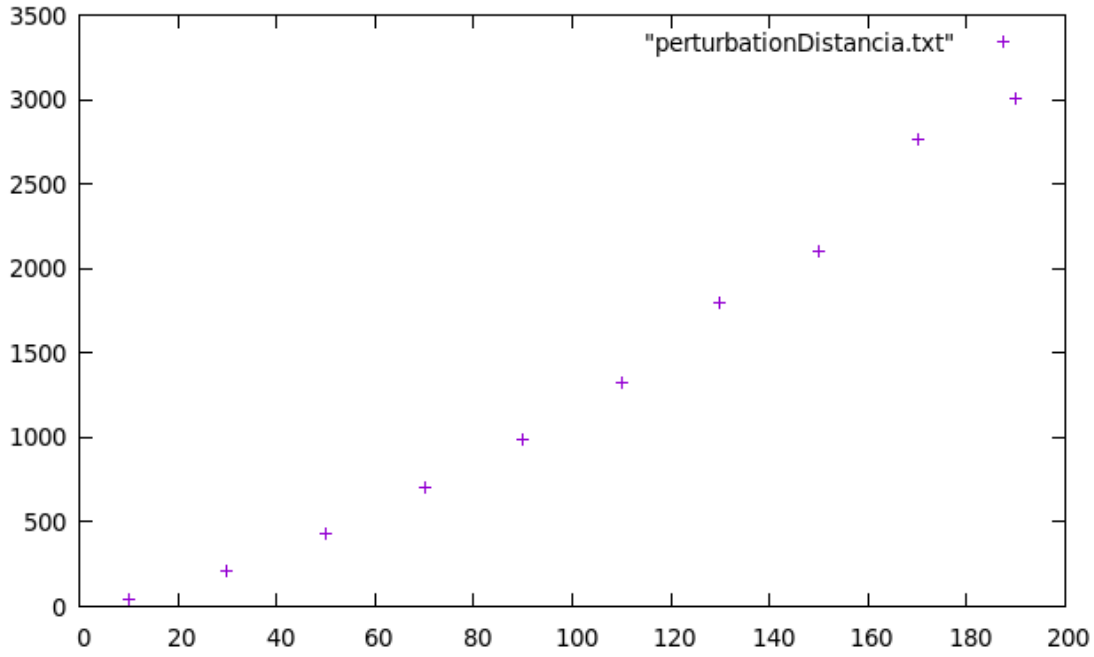


Gráfico 1.3.2. Datos empíricos para viajante de comercio versión perturbaciones, distancia



Los datos de las gráficas se encuentran en la ***sección IV: Anexo II*** de este documento.

1.4 Comparación de enfoques

Los cuatro algoritmos que tenemos son: cercanía, inserción, fuerza bruta y perturbaciones.
Vamos a comparar por separado las gráficas de las distancias y los tiempos de los tres primeros algoritmos.
Como el algoritmo de fuerza bruta solo funciona para tamaños muy pequeños pondremos la cota en 10 nodos.

Gráfico 1.4.1. Contraste de datos empíricos: tiempo

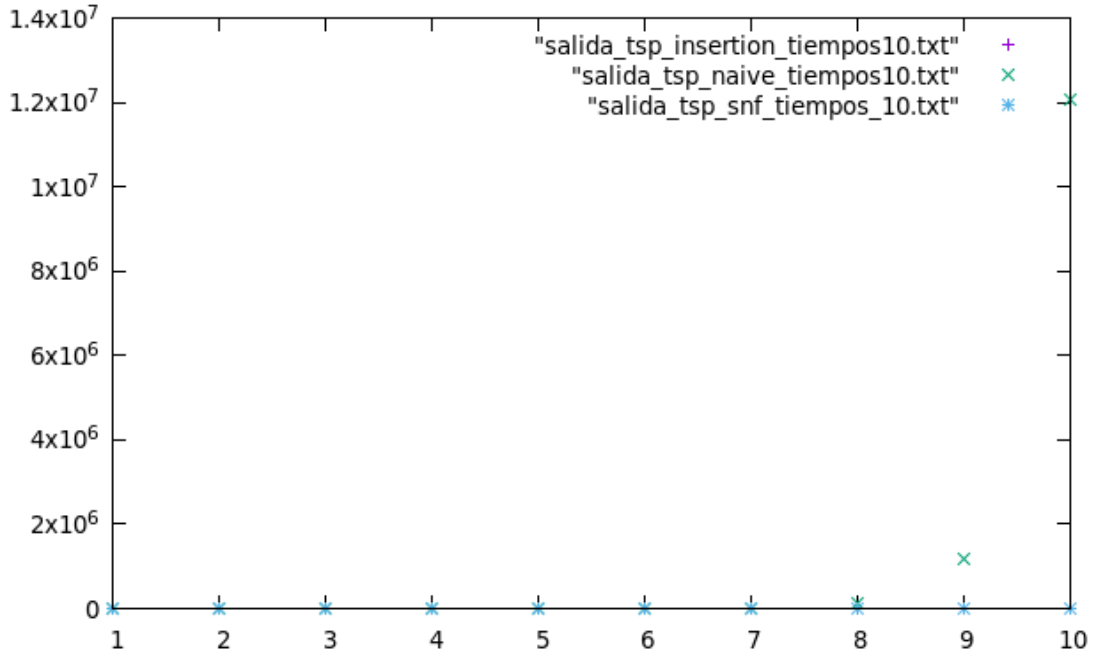
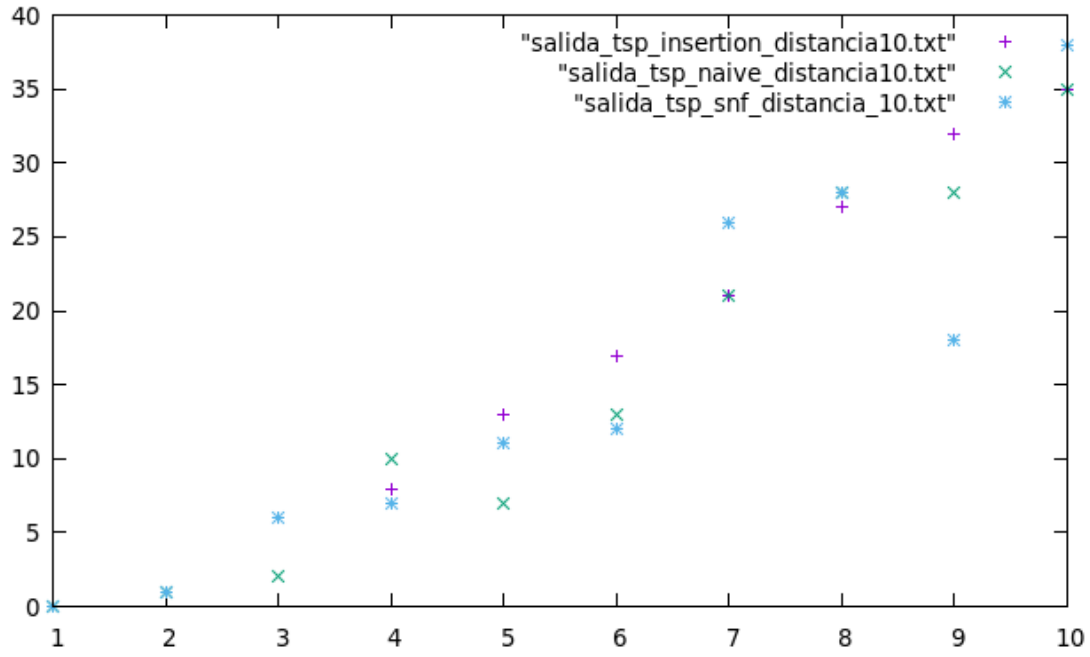


Gráfico 1.4.2. Contraste de datos empíricos: distancia



Podemos ver que las distancias de los tres algoritmos son similares, pero el tiempo que se emplea en obtenerlas es mucho mayor en fuerza bruta. Estudiaremos por tanto los algoritmos *inserción* y *cercanía* por separado.

Gráfico 1.4.3. Contraste de datos empíricos (cercanía e inserción): tiempo

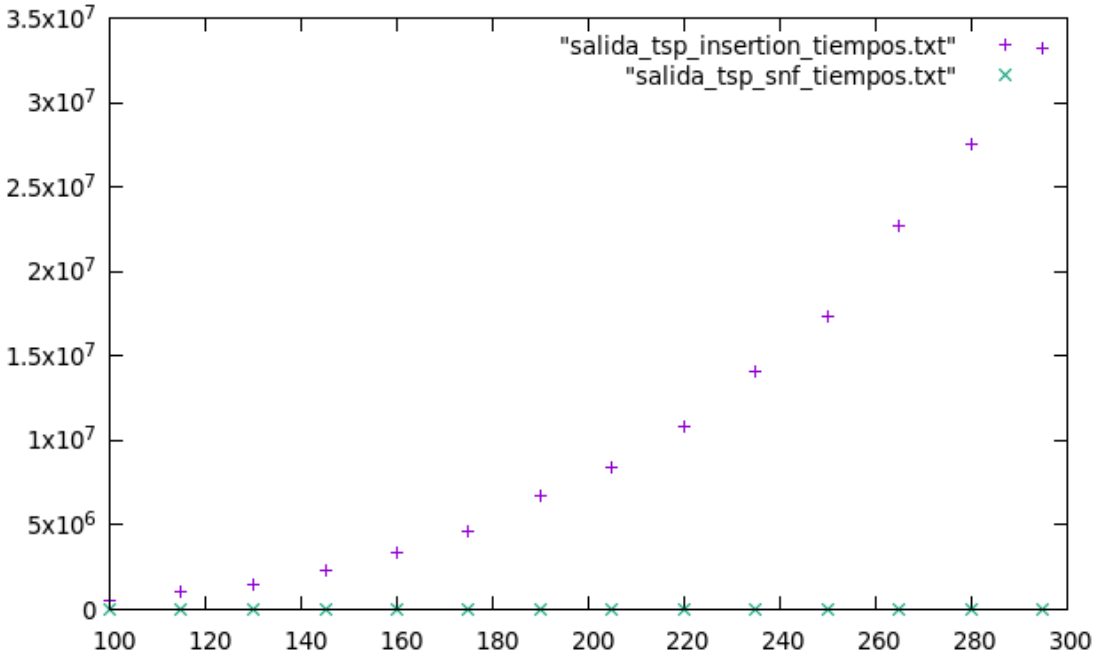
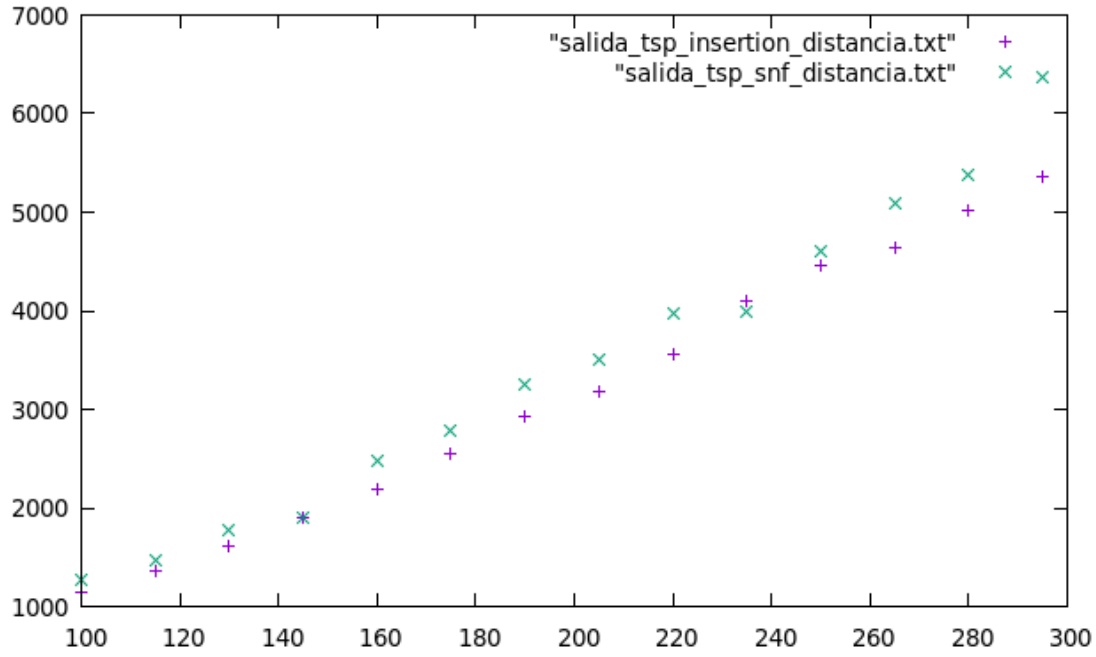


Gráfico 1.4.4. Contraste de datos empíricos (cercanía e inserción): distancia



Observamos que las distancias obtenidas con *inserción* son ligeramente mejores, pero al comparar los tiempos *cercanía* es considerablemente más rápido. Por lo general nos interesará más usar este algoritmo, ya que conseguimos resultados parecidos en un tiempo mucho menor.

Vamos a comparar ahora el algoritmo *cercanía* con nuestro algoritmo: *perturbaciones*

Gráfico 1.4.5. Contraste de datos empíricos (cercanía y perturbaciones): tiempo

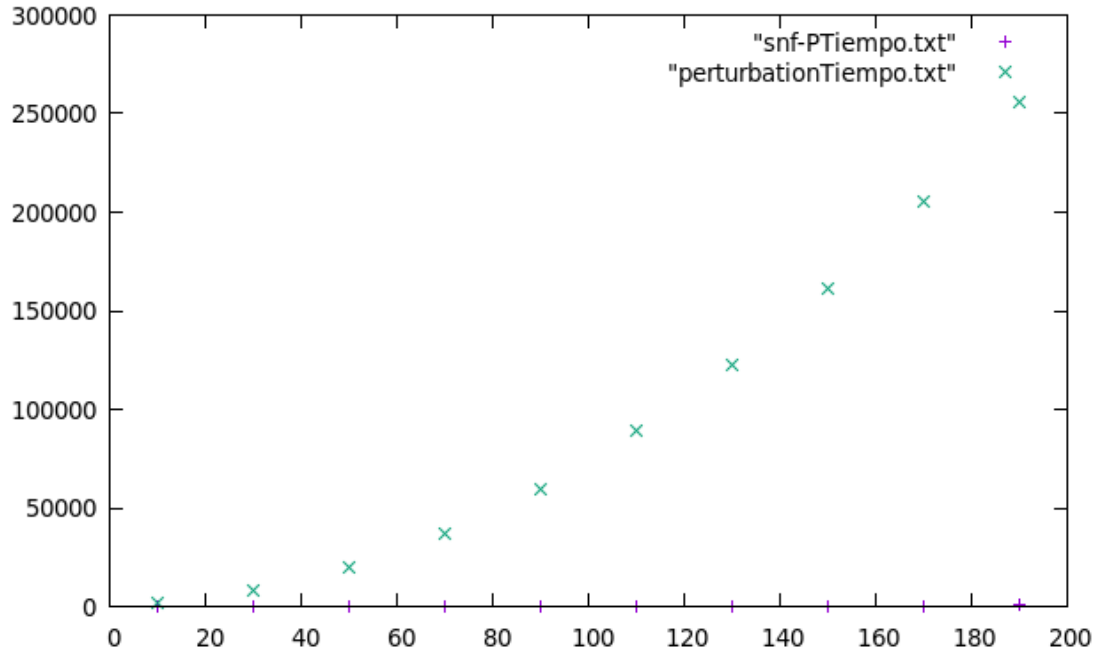
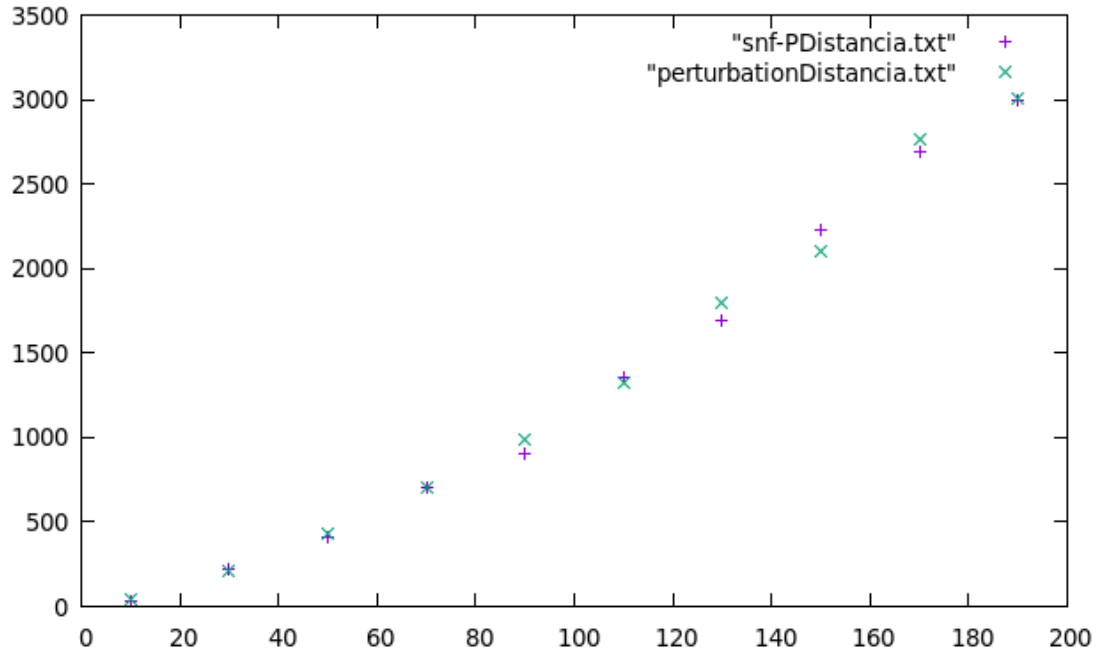


Gráfico 1.4.6. Contraste de datos empíricos (cercanía y perturbaciones): distancia



Observar que la diferencia de tiempo de nuestro algoritmo es considerable respecto a la de *cercanía* y sin embargo la distancia obtenida es similar. Podemos afirmar por tanto que el algoritmo *cercanía* es mejor que el nuestro.

2 Asignación de tareas (*worker*)

> 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.

Elementos comunes

A lo largo de la solución del problema 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:

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

2.1 Enfoque greedy simple (por inserción)

Para distribuir las tareas, relizaremos los siguientes pasos:

1. Recorremos la matriz C por filas, y asignamos al primer trabajador la tarea de menor coste.
2. Inhabilitamos la columna haciendo uso de un vector auxiliar –que hemos llamado d– de trabajos disponibles. También podríamos sobreescribir la matriz C colocando todos los componentes de esa misma columna a 0 –pues los costes son siempre estrictamente positivos–.
3. Realizamos de nuevo este procedimiento hasta que todos los trabajadores tengan una tarea asignada, con la salvedad de que en el paso (2) hemos de comprobar si la tarea está disponible –comprobando el vector de trabajos disponibles o si los elementos no son ceros, en caso de usar la otra alternativa–.

(1)

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

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

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

(2)

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

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

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

(3)

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

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

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

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

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

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

Visualización de los pasos anteriormente descritos

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

El código completo se encuentra en la ***sección IV: Anexo I*** de este documento.

Veremos la eficiencia empírica de este algoritmo.

Análisis empírico

Los tamaños de prueba para ejecutar el algoritmo han ido desde 100 hasta 4000, cada vez con incremento de 200. A su vez cada iteración la hemos hecho 100 veces y hemos calculado la media, con el fin de eliminar los mejores y peores casos.

Los datos obtenidos, calculando por un lado el tiempo y por otro la distancia han sido estos:

Gráfico 2.1.1. Datos empíricos para asignación de tareas, inserción: tiempo

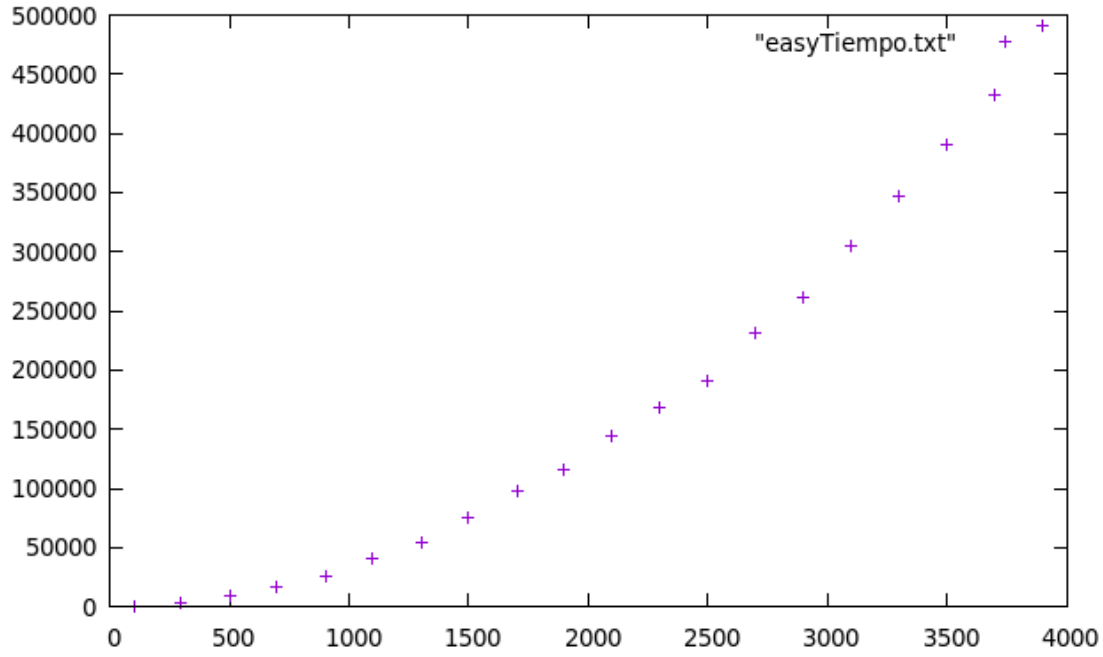
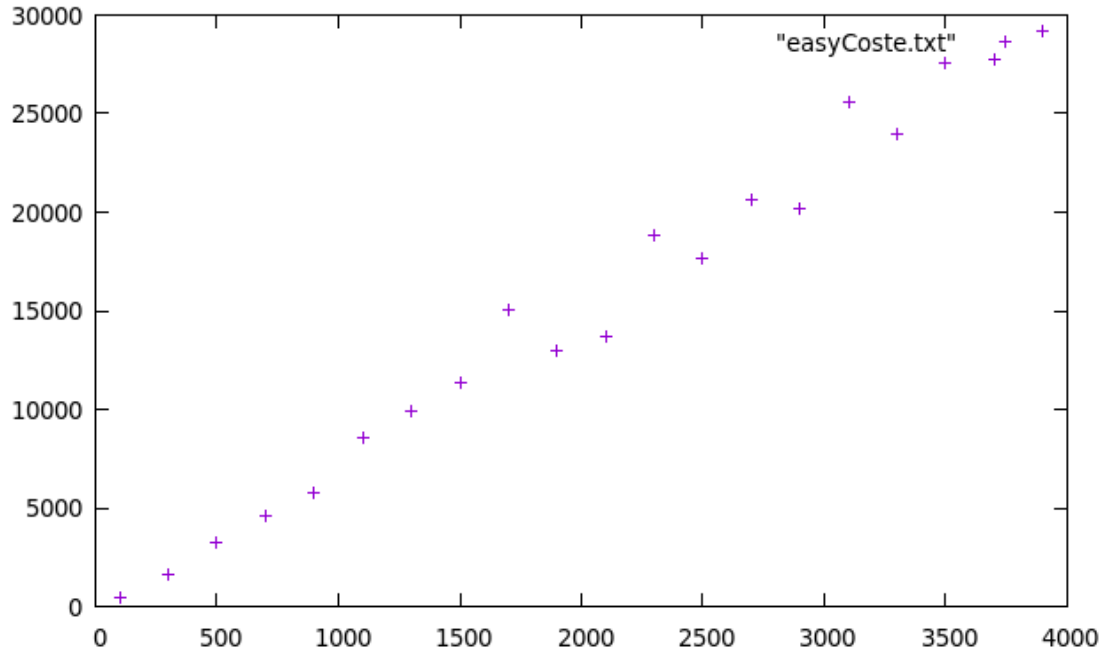


Gráfico 2.1.2. Datos empíricos para asignación de tareas, inserción: coste



Los datos de las gráficas se encuentran en la ***sección IV: Anexo II*** de este documento.

2.2 Enfoque greedy con permutaciones

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

El procedimiento es sencillo:

1. Buscamos el elemento que tiene un coste mayor de todas las asignaciones, al que llamaremos a_m, con m la posición de dicho elemento.
2. Vamos intercambiando los elementos del vector de asignaciones con el de coste mayor, es decir, obtenemos las asignaciones resultantes de realizar esta permutación:

   $$a'_0 = \{a_m, a_1, ..., a_{m-1}, a_0, a_{m+1}, ..., a_{n-1}\}, a'_1 = \{a_0, a_m, a_2, ..., a_{m-1}, a_1, a_{m+1}, a_{n-1}\}, ...$$

   De entre todas las posibilidades, tomaremos la que tiene mayor ganancia, es decir, el p con el $W_{a'_p}$ menor, la que nos proporciona una asignación con menor coste.
3. Repetimos este proceso un número de veces arbitrario, en cada iteración obtendremos una solución mejor, o la misma solución (en cuyo caso no podremos mejorar esta asignación realizando permutaciones).

Podríamos partir de la asignación que proporciona el enfoque anterior, mejorándola con estas 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. Esto se traduce en nuestra implementación a simplemente buscar el máximo del vector a, de asignaciones (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.

El código completo se encuentra en la ***sección IV: Anexo I*** de este documento. En dicho código se parte de las funciones del enfoque anterior: `get_best_solution` (que ha sido renombrado a `get_insertion_solution`) y `find_best_task`.

Veremos la eficiencia empírica de este algoritmo.

Análisis empírico

Los tamaños de prueba para ejecutar el algoritmo han ido desde 1 hasta 100, cada vez con incremento de 10. A su vez cada iteración la hemos hecho 100 veces y hemos calculado la media, con el fin de eliminar los mejores y peores casos.

Los datos obtenidos, calculando por un lado el tiempo y por otro la distancia han sido estos:

Gráfico 2.2.1. Datos empíricos para asignación de tareas, perturbaciones: tiempo

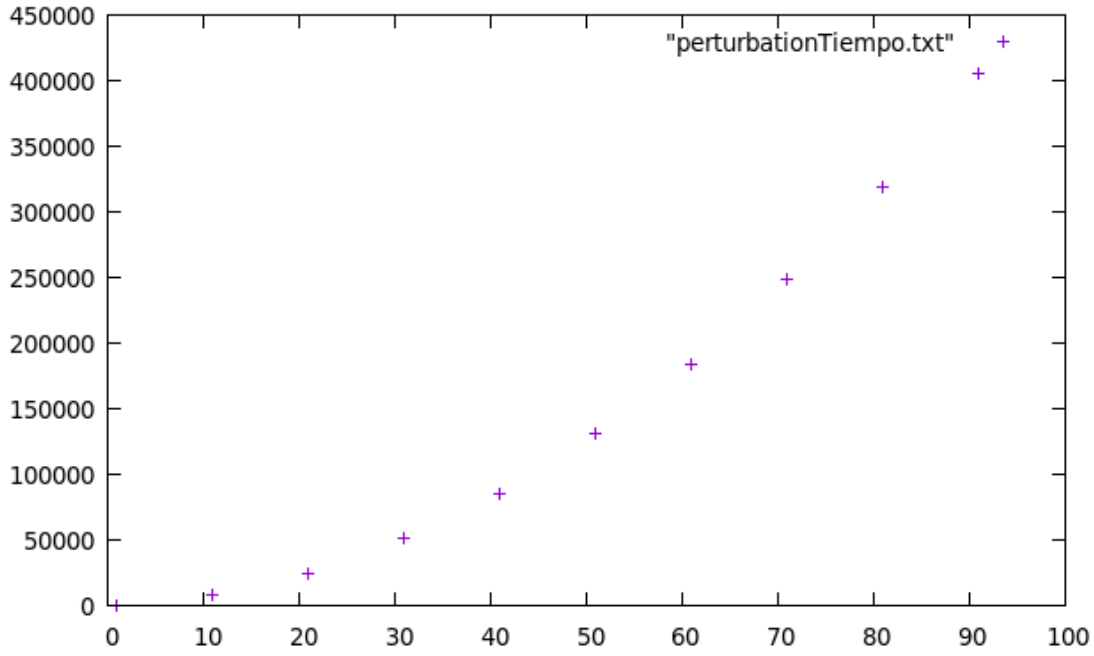
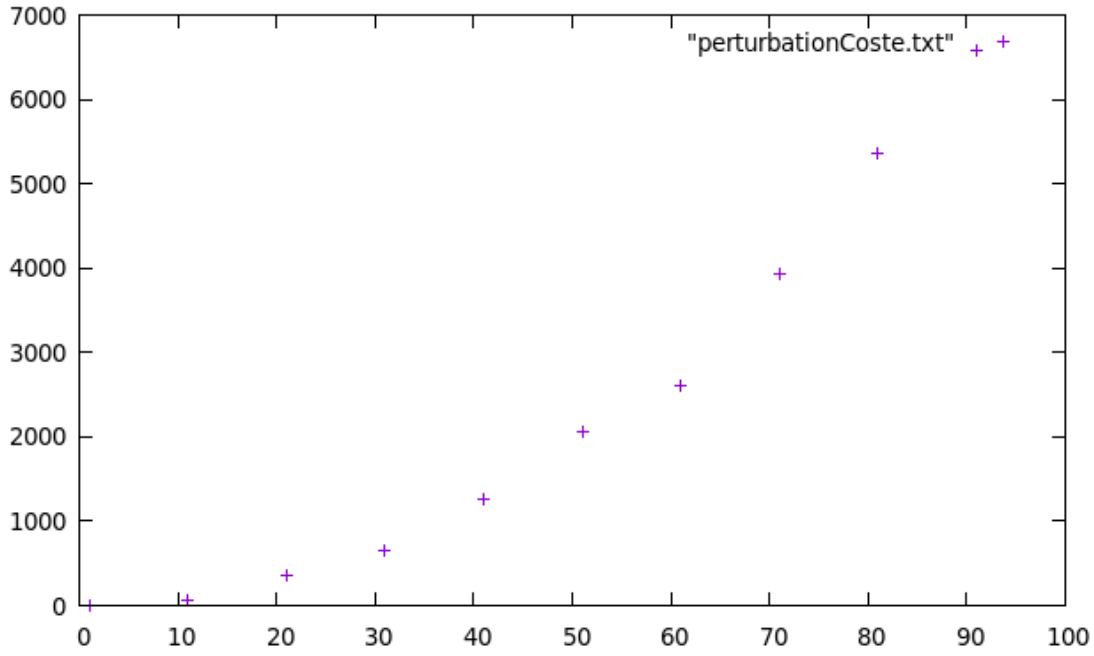


Gráfico 2.2.2. Datos empíricos para asignación de tareas, perturbaciones: coste



Los datos de las gráficas se encuentran en la ***sección IV: Anexo II*** de este documento.

2.3 Comparación de enfoques

Ambos enfoques son *greedy*, e intentan acercarse a la mejor solución. Evidentemente, el primero de ellos, por inserción, no se acerca a la mejor solución de todas. Basta tomar como ejemplo la siguiente matriz de costes:

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

Éste efectuaría una asignación que tendría un coste de 11, mientras que la asignación óptima para este caso tendría un coste de 4.

Para poder resolver esto, el enfoque con permutaciones es una solución muy interesante. Éste solucionaría nuestro problema en este caso, llegando a la asignación óptima tras una sola permutación.

Además, el enfoque con permutaciones es especialmente interesante cuando partimos de una cantidad muy grande de datos, ya que tener una estimación inicial puede ser complicado. Podríamos partir de una asignación cualquiera, y aplicar estas permutaciones de forma sucesiva hasta encontrar asignaciones mejores con cada iteración. En resumen, podemos dejar a nuestro algoritmo calcular asignaciones consecutivamente mejores, simplemente con dejar un tiempo de ejecución mayor.

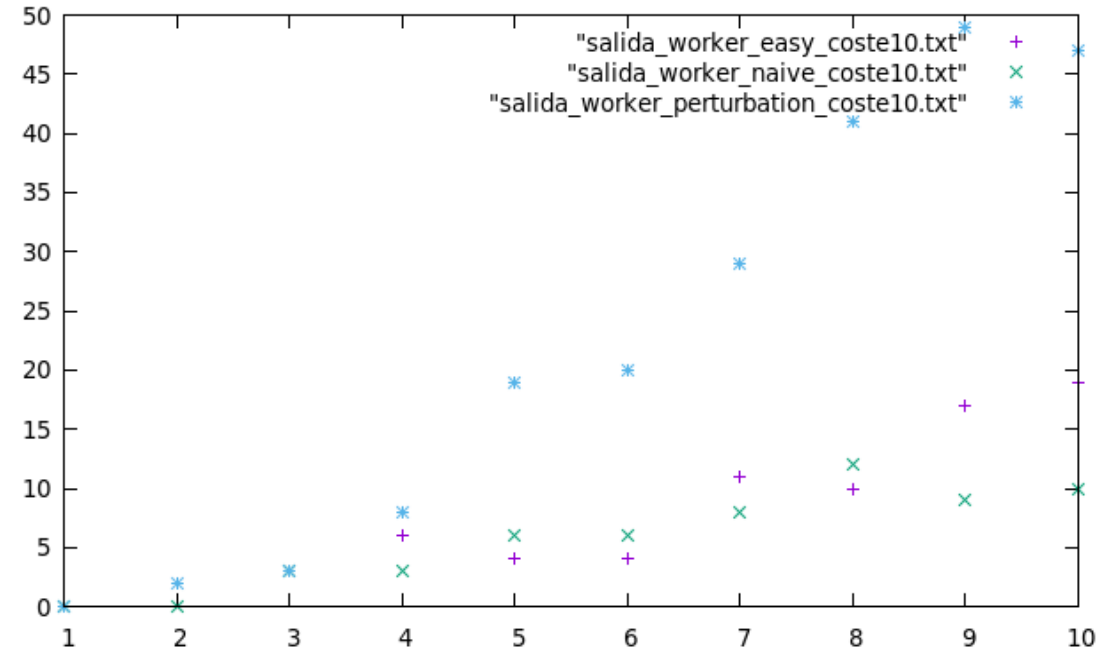
Vemos por tanto cómo podemos, a partir de *greedy*, obtener algoritmos que nos **mejoran** nuestras soluciones conforme van actuando de forma sucesiva.

Comparación en análisis empírico

Comparamos empíricamente los dos algoritmos propuestos y otro hecho por fuerza bruta. Como este último solo funciona para tamaños muy pequeños situamos la cota en 10.

Gráfico 2.3.1. Contraste de datos empíricos: tiempo

Gráfico 2.3.2. Contraste de datos empíricos: coste



Vemos que los mayores valores de coste los obtenemos con *perturbations* mientras que con *fuerza bruta* y *easy* se obtienen valores similares. Para el tiempo, sin embargo, los valores de *fuerza bruta* se disparan. Vamos a estudiar por tanto a más largo plazo los dos primeros algoritmos.

Gráfico 2.3.3. Contraste de datos empíricos: tiempo

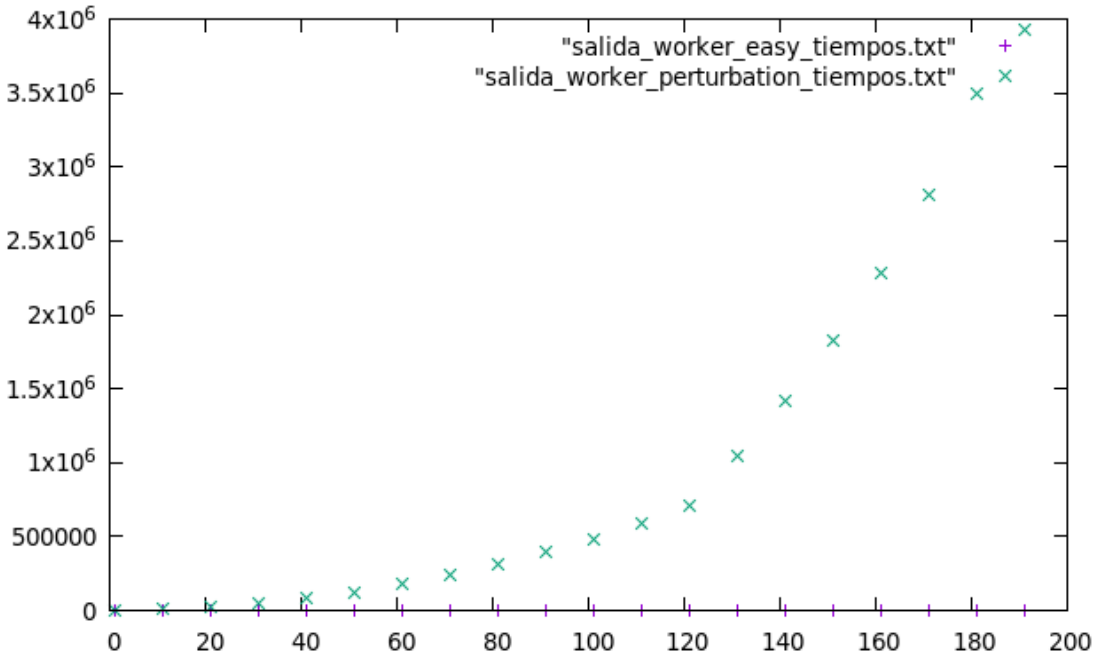
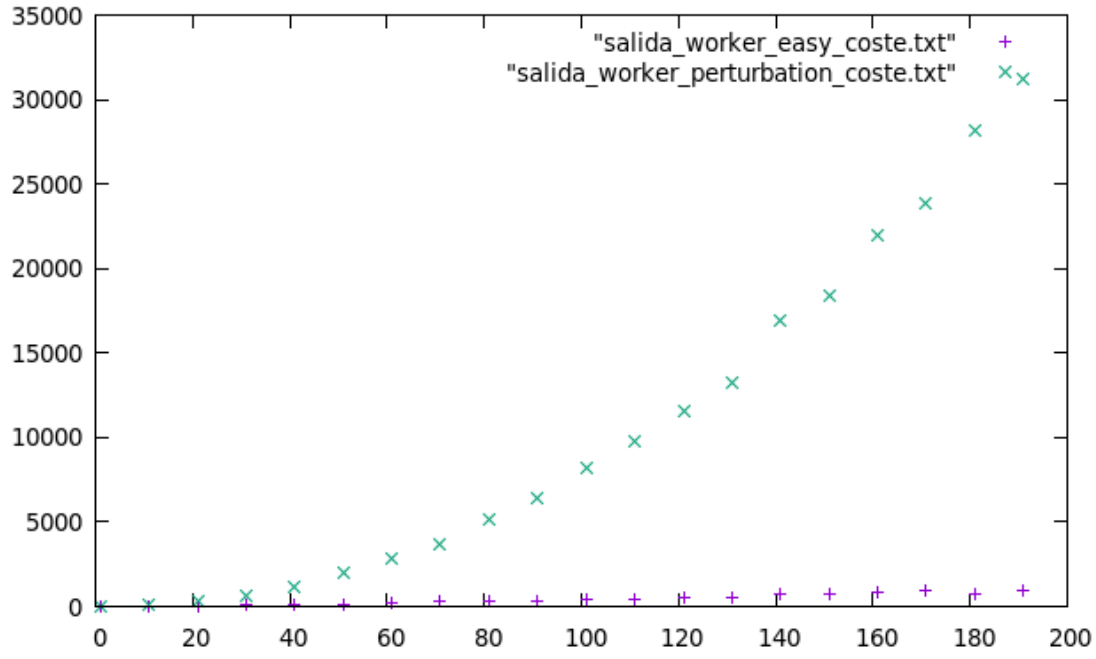


Gráfico 2.3.4. Contraste de datos empíricos: coste



Observar que la gráfica de *perturbations* crece más rápida tanto en tiempo que podemos decir que el algoritmo *easy* es mejor.

III | Conclusiones

Con esta práctica, hemos aprendido a crear algoritmos voraces para resolver problemas que, en su versión en fuerza bruta, tienen una complejidad muy elevada.

Hemos contemplado, dentro de esta técnica, diversos enfoques para cada algoritmo, prestando atención a la dificultad de obtener la solución óptima, y primando el "acercarnos" a ella mediante algoritmos que, con sucesivas iteraciones, van mejorando la solución anterior, hasta llegar a soluciones cada vez más óptimas.

Esto es algo de especial relevancia al trabajar con cantidades ingentes de datos, en el que dar una solución inicial puede ser muy complicada, pudiendo partir incluso de soluciones arbitrarias, mejorándolas conforme nuestros algoritmos se van ejecutando.

IV | Anexos

Anexo I. Códigos

Códigos para TSP

`tsp_common.hpp`

Biblioteca de **funciones comunes** a todos los códigos usados.

```cpp
#include <vector>
#include <stdlib.h>
#include <time.h>
#include <cstdlib>

namespace common{

// Generacion de numeros aleatorios
//=============================================================================

/**
 * @brief Devuelve el valor absoluto de un valor double
 * @param val el double con el que trabajamos
 * @return el valor absoluto de un valor double
 * */
double abs(double val){
    return val < 0 ? - val : val;
}

void startRandom(){
    std::srand(time(NULL));
}

int randomInt(int min, int max){
    int value = min + std::rand()  % (max +1 - min) ;
    return value;
}

double random_double(double min, double max){
    return (std::rand() / (double) RAND_MAX ) * (max - min) + min;
}

// Estructura basicas Punto
//=============================================================================
/**
 * @brief Estructura para representar un punto en el plano
 *
 * En esta estructura se implementa la distancia base que usa el resto de las funciones
 * */
```

```cpp
struct Point{
    double x = 0;
    double y = 0;

    /**
     * @brief Distancia entre dos puntos
     *
     * Cambiar esta funcion si queremos cambiar la distancia entre dos puntos para
     * cualquier funcion que haga uso de la distancia (esta es la implementacion base)
     * */
    double distance(Point other){
        return abs(other.x - x) + abs(other.y - y);
    }
};

// Generacion de datos para el problema
//================================================================================

/**
 * @brief Genera los puntos necesarios para el problema
 * @param num_points, la cantidad de puntos con la que queremos trabajar
 * @return un vector de puntos, cuyas coordenadas estan en el intevalo [0, num_points]
 * */
std::vector<Point> generate_problem_data(int num_points){
    std::vector<Point> points;

    for(int i = 0; i < num_points; i++){
        double x = random_double(0, num_points);
        double y = random_double(0, num_points);

        Point p = {x, y};
        points.push_back(p);
    }

    return points;
}

// Funcionalidades para solucionar partes comunes del problema
//================================================================================

/**
 * @brief Genera un camino dado una permutacion de indices
 * @param points, los puntos sobre los que queremos calcular el camino
 * @param permutation, la permutacion de indices
 * @return un camino dado los puntos y la permutacion
 * */
std::vector<Point> generate_road_by_indexes(std::vector<common::Point> points, std::vector<int
    > permutation){
    std::vector<Point> road;

    for(auto index : permutation){
        road.push_back(points[index]);
    }

    return road;
```

```
}

/**
 * @brief Genera el camino resultado de intercambiar dos posiciones de un camino
 * @param road, el camino base
 * @param pos1, la primera posicon con la que intercambiamos
 * @param pos2, la segunda posicon con la que intercambiamos
 * @return un camino resultado del cambio descrito
 * */
std::vector<Point> get_swap(std::vector<Point> road, int pos1, int pos2){
    std::vector<Point> changed = road;

    Point tmp = road[pos1];
    road[pos1] = road[pos2];
    road[pos2] = tmp;

    return changed;
}

// Calculo de distancias
//================================================================================
/**
 * @brief Calcula la distancia entre dos puntos
 * @param p1, el primer punto del que queremos calcular la distancia
 * @param p2, el segundo punto del que queremos calcular la distancia
 * @return al distancia entre ambos puntos
 * */
double distance(Point p1, Point p2){
    return p1.distance(p2);
}

/**
 * @brief Calcula la distancia de un camino dado
 * @param road, vector de puntos que forman un camino
 * @return la distancia total de ese camino
 * */
double get_road_distance(std::vector<Point> road){
    double road_distance = 0;

    // Distancia entre los n puntos
    for(int i = 0; i < road.size() - 1; i++){
        road_distance = road_distance + distance(road[i], road[i+1]);

    }

    // Distancia entre el ultimo y primer punto
    road_distance = road_distance + distance(road[0], road[road.size()-1]);

    return road_distance;
}

} // namespace common
```

tsp_naive.cpp

Resolución del algoritmo por **fuerza bruta**.

```cpp
#include <vector>
#include <string>
#include <iostream>
#include <algorithm>
#include "tsp_common.hpp"
using namespace std;

// Declaracion de las funciones que vamos a calcular
//================================================================================

/**
 * @brief Genera todas las permutaciones posibles del conjunto {0, 1, ..., n-1}
 * @return un vector con todas las permutaciones posibles
 * */
vector<vector<int> > generate_permutations(int n);

/**
 * @brief Calcula la mejor solucion para un conjunto de puntos, por fuerza bruta
 * @param points, el conjunto de puntos sobre el que vamos a operar
 * @return el camino cuya distancia es la minima
 * */
vector<common::Point> get_best_solution(vector<common::Point> points);



int main(){
    // Parametro del programa para probar cosas
    int num_points = 10;

    // Puntos con los que voy a trabajar
    vector<common::Point> points = common::generate_problem_data(10);

    // Soluciono el problema por fuerza bruta
    vector<common::Point> road = get_best_solution(points);

    // Muestro el resultado del problema
    cout << "El camino optimo para el problemma es: " << endl;
    for(auto point : road){
        cout << "x: " << point.x << " y: " << point.y << endl;
    }

    cout << "La distancia del camino optimo es: " << get_road_distance(road) << endl;


    // Todo ha salido OK
    return 0;
}

// Implementacion de las funciones
//================================================================================
vector<vector<int> > generate_permutations(int n){
    vector<vector<int> > permutations;
```

```
53
54      // We get the base permutation to work with
55      vector<int> base_perm;
56      for(int i = 0; i < n; i++){
57          base_perm.push_back(i);
58      }
59
60      do{
61          permutations.push_back(base_perm);
62      }while(next_permutation(base_perm.begin(), base_perm.end()));
63
64
65      return permutations;
66  }
67
68  vector<common::Point> get_best_solution(vector<common::Point> points){
69      // Permutaciones con las que vamos a trabajar
70      vector<vector<int> > permutations = generate_permutations(points.size());
71
72      // Camino y distancia minima
73      vector<common::Point> min_road = points;
74      double min_distance = common::get_road_distance(min_road);
75
76      // Calculo las distancias de todas las permutaciones
77      for(auto permutation : permutations){
78          vector<common::Point> current_road = common::generate_road_by_indexes(points,
        permutation);
79          double current_distance = common::get_road_distance(current_road);
80
81          if(current_distance < min_distance){
82              min_distance = current_distance;
83              min_road = current_road;
84          }
85      }
86
87      return min_road;
88  }
```

`tsp_matrix.cpp`

Resolución usando **matrices de distancias**.

```
1  #include <iostream>
2  #include <vector>
3  #include "tsp_common.hpp"
4  using namespace std;
5
6  // Declaracion de funciones con las que trabajamos
7  //================================================================================
8
9  /**
10  * @brief Calcula la mejor solucion para un conjunto de puntos, por Shortest Neighbor First
11  * @param points, el conjunto de puntos sobre el que vamos a operar
12  * @pre points.size() > 0
13  * @return el camino cuya distancia es la minima, si todo sale bien
```

```
 *         un vector vacio, si ocurre algun error
 * */
vector<common::Point> get_best_solution(vector<common::Point> points);

/**
 * @brief Genera una matriz con las distancias entre los puntos
 * @param points, los puntos sobre los que calculamos las distancias
 * @return una matriz con las distancias entre los puntos
 *
 * Matriz[i][j] := distance(pi, pj)
 *
 *
 * WIP -- Tiene una violacion del segmento
 * */
vector<vector<double> > generate_distances_matrix(vector<common::Point> points);

/**
 * @brief Limpia las distancias de una determinada posicion
 *        Ello implica poner a cero la fila y columna de una determinada posicion
 * @param distances_matrix, matriz sobre la que trabajamos, SE MODIFICA
 * @param pos, la posicion sobre la que tenemos que limpiar
 * */
void clean_position(vector<vector<double> > & distances_matrix, int pos);

/**
 * @brief Halla la posicion en una fila con la menor distancia
 * @param distances_matrix, la matriz con las distancias
 * @param position, la posicion respecto la que queremos buscar el minimo elemento
 * @return la posicion cuya distancia es la minima, si todo sale bien
 *         -1, si ocurre algun error
 * */
int get_min_row_element(vector<vector<double> > distances_matrix, int position);

/**
 * @brief Genera una matriz size x size llena de ceros
 * @parma size, la dimension de la matriz cuadrada
 * @return la matriz cuadrada llena de ceros
 * */
vector<vector<double> > get_empty_matrix(int size);

// Funcion principal
//=============================================================================
int main(){
    // Parametro sobre el size del problema
    int problem_size = 100;

    // Inicio los numeros aleatorios
    common::startRandom();

    // Puntos con los que vamos a trabajar
    vector<common::Point> points = common::generate_problem_data(problem_size);

    // Calculo la solucion aproximada
    vector<common::Point> road = get_best_solution(points);

```

```
69      // Muestro el resultado
70      cout << "La distancia de la solucion obtenida es: " << common::get_road_distance(road) <<
        endl;
71
72      // Todo ha salido OK
73      return 0;
74  }
75
76  // Implementacion de las funciones
77  //=========================================================================
78  void clean_position(vector<vector<double> > & distances_matrix, int pos){
79      // Cleaning the row
80      for(int col = 0; col < distances_matrix.size(); col++){
81          distances_matrix[pos][col] = 0;
82      }
83
84      // Cleaning the col
85      for(int row = 0; row < distances_matrix.size(); row++){
86          distances_matrix[row][pos] = 0;
87      }
88  }
89
90  int get_min_row_element(vector<vector<double> > distances_matrix, int position){
91      int min_pos = 0;
92      int starting_pos = 0;
93
94      // Busco la primera posicion con distancia no nula
95      while(distances_matrix[position][starting_pos] == 0 && starting_pos < distances_matrix.
        size()){
96          starting_pos = starting_pos + 1;
97      }
98
99      // Comprobacion de seguridad
100     if(starting_pos = distances_matrix.size()-1){
101         cerr << "ERROR! Posicion no encontrada en get_min_row_element()" << endl;
102         cerr << "Se devuelve -1 como codigo de error!" << endl;
103         return -1;
104     }
105
106     // Busco el elemento optimo de la columna
107     min_pos = starting_pos;
108     double min_val = distances_matrix[position][starting_pos];
109
110     for(int col = starting_pos; col < distances_matrix[position].size(); col++){
111         if(distances_matrix[position][col] < min_val){
112             min_pos = col;
113             min_val = distances_matrix[position][col];
114         }
115     }
116
117     return min_pos;
118 }
119
120 vector<common::Point> get_best_solution(vector<common::Point> points){
121     vector<common::Point> road;
```

```
122     vector<vector<double> > distances_matrix = generate_distances_matrix(points);
123 
124     // WIP
125     for(int row = 0; row < distances_matrix.size(); row++){
126         for(int col = 0; col < distances_matrix.size(); col++){
127             cout << distances_matrix[row][col] << " ";
128         }
129         cout << endl;
130     }
131 
132     // Empiezo siempre por el primer punto
133     int last_point = 0;
134     road.push_back(points[0]);
135 
136     while(road.size() < points.size()){
137         // Hallo el punto mas cercano al anterior
138         int best_position = get_min_row_element(distances_matrix, last_point);
139 
140         // Comprobacion de seguridad
141         if(best_position == -1){
142             cerr << "ERROR! Posicion optima no encontrada en get_best_solution()" << endl;
143             cerr << "Se devuelve un vector nulo!" << endl;
144 
145             // Devuelvo un vector vacio
146             return vector<common::Point>();
147         }
148 
149         // Lo inserto al road
150         road.push_back(points[best_position]);
151 
152         // Limpio la fila y columna del anterior punto, ya no me hace falta
153         clean_position(distances_matrix, best_position);
154 
155         // Tomo el nuevo punto anterior
156         last_point = best_position;
157     }
158 
159     return road;
160 }
161 
162 vector<vector<double> > generate_distances_matrix(vector<common::Point> points){
163     // Genero una matriz de points.size() x points.size() llena de ceros
164     vector<vector<double> > distances_matrix = get_empty_matrix(points.size());
165 
166     for(int row = 0; row < points.size(); row++){
167         for(int col = 0; col < points.size(); col++){
168             distances_matrix[row][col] = common::distance(points[row], points[col]);
169         }
170     }
171 
172     return distances_matrix;
173 }
174 
175 vector<vector<double> > get_empty_matrix(int size){
176     vector<vector<double> > empty;
```

```
177     vector<double> empty_row(size, 0);
178 
179     for(int i = 0; i < size; i++){
180         empty.push_back(empty_row);
181     }
182 
183     return empty;
184 }
```

tsp_snf.cpp

Solución por **SNF** usando un vector con los puntos que quedan.

```
1  #include <iostream>
2  #include <vector>
3  #include "tsp_common.hpp"
4  using namespace std;
5  
6  // Declaracion de funciones con las que trabajamos
7  //============================================================================
8  
9  /**
10  * @brief Calcula la mejor solucion para un conjunto de puntos, por Shortest Neighbor First
11  * @param points, el conjunto de puntos sobre el que vamos a operar
12  * @pre points.size() > 0
13  * @return el camino cuya distancia es la minima
14  * */
15 vector<common::Point> get_best_solution(vector<common::Point> points);
16 
17 // Funcion principal
18 //============================================================================
19 int main(){
20     // Parametro sobre el size del problema
21     int problem_size = 10000;
22 
23     // Inicio los numeros aleatorios
24     common::startRandom();
25 
26     // Puntos con los que vamos a trabajar
27     vector<common::Point> points = common::generate_problem_data(problem_size);
28 
29     // Calculo la solucion aproximada
30     vector<common::Point> road = get_best_solution(points);
31 
32     // Muestro el resultado
33     cout << "La distancia de la solucion obtenida es: " << common::get_road_distance(road) <<
       endl;
34 
35     // Todo ha salido OK
36     return 0;
37 }
38 
39 // Implementacion de las funciones
40 //============================================================================
41 vector<common::Point> get_best_solution(vector<common::Point> points){
```

```
    vector<common::Point> road;                 // Solucion que vamos a construir
    vector<common::Point> points_left = points; // Puntos que quedan por insertar a la
    solucion

    // Parto siempre del primer punto del vector
    road.push_back(points_left[0]);
    points_left.erase(points_left.begin() + 0);

    // Voy construyendo la solucion, sacando puntos de points_left y colocandolos en road
    while(points_left.size() > 0){
        double min_distance = common::distance(road[road.size() -1], points_left[0]);
        int min_pos = 0;

        // Buscamos el punto mas cercano
        for(int i = 0; i < points_left.size(); i++){
            double current_distance = common::distance(road[road.size() -1], points_left[i]);
            if(current_distance < min_distance){
                min_distance = current_distance;
                min_pos = i;
            }
        }

        // Insertamos el punto mas cercano a la solucion y lo quitamos de los puntos que
    faltan
        road.push_back(points_left[min_pos]);
        points_left.erase(points_left.begin() + min_pos);
    }

    return road;
}
```

`tsp_perturbation.cpp`

Solución usando **perturbaciones**, partiendo del SNF anterior.

```
#include <iostream>
#include <vector>
#include "tsp_common.hpp"
using namespace std;

// Declaracion de funciones con las que trabajamos
//================================================================================

/**
 * @brief Calcula la solucion para un conjunto de puntos, por Shortest Neighbor First
 * @param points, el conjunto de puntos sobre el que vamos a operar
 * @pre points.size() > 0
 * @return el camino siguiendo este algoritmo
 * */
vector<common::Point> get_snf_solution(vector<common::Point> points);

/**
 * @brief Calcula un camino partiendo de un algoritmo SNF y aplicandole una serie de
    perturbaciones
 * @param points, conjunto de puntos sobre el que calcular el mejor camino posible
```

```
 * @parma perturbations, numero de perturbaciones que se van a realizar
 * @return el camino obtenido como ya se ha descrito
 * */
vector<common::Point> get_best_solution(vector<common::Point> points, int perturbations);

/**
 * @brief Perturba un camino respecto de un punto dado
 * @brief road, el camino a perturbar, SE MODIFICA
 * @param pos, la posicion respecto la que se perturba
 * */
void perturbate(vector<common::Point> road, int pos);

/**
 * @brief Calcula el punto cuya distancia al siguiente punto es mayor
 * @param road, camino cuyo peor camino hay que calcular
 * @return la posicion del peor punto ya descrito
 * */
int get_worst_node(vector<common::Point> road);

// Funcion principal
//============================================================================
int main(){
    // Parametro sobre el size del problema
    int problem_size = 100;
    int perturbations = 100;

    // Inicio los numeros aleatorios
    common::startRandom();

    // Puntos con los que vamos a trabajar
    vector<common::Point> points = common::generate_problem_data(problem_size);

    // Calculo la solucion aproximada
    vector<common::Point> road = get_best_solution(points, perturbations);

    // Muestro el resultado
    cout << "La distancia de la solucion obtenida es: " << common::get_road_distance(road) <<
    endl;

    // Todo ha salido OK
    return 0;
}

// Implementacion de las funciones
//============================================================================
vector<common::Point> get_snf_solution(vector<common::Point> points){
    vector<common::Point> road;                 // Solucion que vamos a construir
    vector<common::Point> points_left = points; // Puntos que quedan por insertar a la
    solucion

    // Parto siempre del primer punto del vector
    road.push_back(points_left[0]);
    points_left.erase(points_left.begin() + 0);

    // Voy construyendo la solucion, sacando puntos de points_left y colocandolos en road
```

```
73      while(points_left.size() > 0){
74          double min_distance = common::distance(road[road.size() -1], points_left[0]);
75          int min_pos = 0;
76
77          // Buscamos el punto mas cercano
78          for(int i = 0; i < points_left.size(); i++){
79              double current_distance = common::distance(road[road.size() -1], points_left[i]);
80              if(current_distance < min_distance){
81                  min_distance = current_distance;
82                  min_pos = i;
83              }
84          }
85
86          // Insertamos el punto mas cercano a la solucion y lo quitamos de los puntos que
        faltan
87          road.push_back(points_left[min_pos]);
88          points_left.erase(points_left.begin() + min_pos);
89      }
90
91      return road;
92  }
93
94  vector<common::Point> get_best_solution(vector<common::Point> points, int perturbations){
95      // Tomo la solucion dada por el snf
96      vector<common::Point> base_road = get_snf_solution(points);
97
98      // Perturbo el numero de veces indicada
99      for(int i = 0; i < perturbations; i++){
100         // Calculo la posicion respecto a la que perturbar
101         int pos = get_worst_node(points);
102
103         // Perturbo el camino base
104         perturbate(base_road, pos);
105     }
106
107     return base_road;
108 }
109
110 void perturbate(vector<common::Point> road, int pos){
111     vector<common::Point> current_perb = road;
112     double best_gain = 0;
113     int best_perturbation = pos;
114     double base_distance = common::get_road_distance(road);
115
116     // Calculo la distancia de todas las permutaciones
117     for(int i = 0; i < road.size(); i++){
118         // Calculo la ganancia de esta permutacion
119         current_perb = common::get_swap(road, pos, i);
120         double swap_distance = common::get_road_distance(current_perb);
121         double current_gain = swap_distance - base_distance;
122
123         // Compruebo si he mejorado la ganancia
124         if(current_gain > best_gain){
125             best_perturbation = i;
126             best_gain = current_gain;
```

```
        }
    }

    // Hago el cambio con la mejor perturbacion
    road = common::get_swap(road, pos, best_perturbation);
}

int get_worst_node(vector<common::Point> road){
    double worst_distance = common::distance(road[0], road[1]);
    int worst_pos = 0;

    // Recorro todos los puntos para ver cual es el peor
    for(int i = 0; i < road.size() - 1; i++){
        if(distance(road[i], road[i+1]) > worst_distance){
            worst_distance = distance(road[i], road[i+1]);
            worst_pos = i;
        }
    }

    // Compruebo la distancia del ultimo al primero
    if(distance(road[road.size()-1], road[0]) > worst_distance){
        worst_distance = road.size() - 1;
    }

    return worst_pos;
}
```

Códigos para *worker*

worker_common.cpp

Biblioteca de **funciones comunes** a todos los códigos usados.

```
#include <iostream>
#include <vector>
#include <stdlib.h>
#include <time.h>
#include <cstdlib>

namespace common{

// Generacion de numeros aleatorios
//================================================================================

/**
 * @brief Devuelve el valor absoluto de un valor double
 * @param val el double con el que trabajamos
 * @return el valor absoluto de un valor double
 * */
double abs(double val){
    return val < 0 ? - val : val;
}

```

```
/**
 * @brief inicia la generacion de numeros aleatorios
 * */
void startRandom(){
    std::srand(time(NULL));
}

/**
 * @brief Genera un double aleatorio en un intervalo dado
 * @param min, el minimo valor que se puede tomar
 * @param max, el maximo valor que se puede tomar
 * @return un double en el intervalo [min, max]
 * */
double random_double(double min, double max){
    return (std::rand() / (double) RAND_MAX ) * (max - min) + min;
}

/**
 * @brief Genera un entero aleatorio en un intervalo dado
 * @param min, el minimo valor que se puede tomar
 * @param max, el maximo valor que se puede tomar
 * @return un entero en el intervalo [min, max]
 * */
double random_int(int min, int max){
    return (int)(std::rand() / (double) RAND_MAX ) * (max - min) + min;
}

// Generacion de los datos del problema
//=============================================================================
/**
 * @brief Genera una matriz aleatoria con las asignaciones de trabajo
 * @param size, size del problema (no. de trabajadores y trabajos)
 * @return una matriz cuadrada de dimension size con valores aleatorios en el intervalo [0,
    size]
 * */
std::vector<std::vector<double> > generate_matrix(int size){
    // Inicio una matriz llena de ceros
    std::vector<double> empty_row(size, 0);
    std::vector<std::vector<double> > matrix(size, empty_row);

    for(int row = 0; row < size; row++){
        for(int col = 0; col < size; col++){
            matrix[row][col] = random_double(0, size);
        }
    }

    return matrix;
}

// Calculos sobre datos del problema
//=============================================================================
/**
 * @brief Calcula el coste de una asignacion dada
 * @param matrix, la matriz con los costes
 * @param asignation, la asignacion cuyo coste se calcula
```

```
 * @return el coste de la asignacion
 * */
double get_cost(std::vector<std::vector<double> > matrix, std::vector<int> asignation){
    double cost = 0;

    for(int i = 0; i < asignation.size(); i++){
        cost = cost + matrix[i][asignation[i]];
    }

    return cost;
}

// Muestra de datos del problema
//=============================================================================
void show_matrix(std::vector<std::vector<double> > matrix, char sep = '\t'){
    for(int row = 0; row < matrix.size(); row++){
        for(int col = 0; col < matrix[row].size(); col++){
            std::cout << matrix[row][col] << sep;
        }
        std::cout << std::endl;
    }
}

}// namespace common
```

worker_naive.cpp

Resolución mediante **fuerza bruta**.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "worker_common.hpp"
using namespace std;

// Declaracion de funciones auxiliares
//=============================================================================
/**
 * @brief Genera todas las permutaciones del conjunto {0, 1, ..., n-1}
 * @param n, el maximo del conjunto sobre el que se permuta
 * @return un array con todas las permutaciones anteriormente descritas
 * */
vector<vector<int> > generate_permutations(int n);

/**
 * @brief Calcula la asignacion optima dada una matriz de costes, usando fuerza bruta
 * @param matrix, la matriz que almacena los costes
 * @return la asignacion optima
 * */
vector<int> get_best_solution(vector<vector<double> > matrix);

// Funcion principal
//=============================================================================
int main(){
    // Parametro para hacer pruebas con las entradas
```

```
    int size = 4;

    // Inicio la generacion de numeros aleatorios
    common::startRandom();

    // Tomo una matriz con los datos del problema
    vector<vector<double> > worker_matrix = common::generate_matrix(size);

    // Calculo la solucion al problema
    vector<int> asignation = get_best_solution(worker_matrix);

    // Muestro el coste de la solucion optima
    cout << "El coste optimo es: " << common::get_cost(worker_matrix, asignation) << endl;

    // Muestro las asignaciones hechas
    for(int i = 0; i < size; i++){
        cout << "Trabajador " << i << " asignado al trabajo " << asignation[i] << endl;
    }

    // Todo ha salido OK
    return 0;
}

// Implementacion de funciones auxiliares
//=============================================================================
vector<vector<int> > generate_permutations(int n){
    vector<vector<int> > permutations;

    // We get the base permutation to work with
    vector<int> base_perm;
    for(int i = 0; i < n; i++){
        base_perm.push_back(i);
    }

    do{
        permutations.push_back(base_perm);
    }while(next_permutation(base_perm.begin(), base_perm.end()));


    return permutations;
}

vector<int> get_best_solution(vector<vector<double> > matrix){
    // Tomo el array de permutaciones, del que me tengo que quedar con solo una
    // permutacion, la permutacion de la asignacion optima
    vector<vector<int> > permutations = generate_permutations(matrix.size());

    // Valores base para empezar a realizar comparaciones
    int best_permutation = 0;
    double best_cost = common::get_cost(matrix, permutations[0]);

    // Comparo todas las permutaciones
    for(int i = 0; i < permutations.size(); i++){
        double current_cost = common::get_cost(matrix, permutations[i]);

```

```
        if(current_cost < best_cost){
            best_cost = current_cost;
            best_permutation = i;
        }
    }

    return permutations[best_permutation];
}
```

worker_easy.cpp

Resolución del problema haciendo uso de **greedy**, en el enfoque simple.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "worker_common.hpp"
using namespace std;

// Declaracion de funciones auxiliares
//=============================================================================
/**
 * @brief Calcula la asignacion parcialmente optima dada una matriz de costes, usando un
    algoritmo greedy
 * @param matrix, la matriz que almacena los costes
 * @return la asignacion parcialmente optima
 * */
vector<int> get_best_solution(vector<vector<double> > matrix);

/**
 * @brief Halla dado un vector de costo de tareas, la mejor tarea que se encuentra disponible
 * @param tasks_cost, vector con los costos de las tareas
 * @param available, vector para controlar si las tareas se encuentra o no disponibles
 * */
int find_best_task(vector<double>tasks_cost, vector<bool> available);

// Funcion principal
//=============================================================================
int main(){
    // Parametro para hacer pruebas con las entradas
    int size = 4000;

    // Inicio la generacion de numeros aleatorios
    common::startRandom();

    // Tomo una matriz con los datos del problema
    vector<vector<double> > worker_matrix = common::generate_matrix(size);

    // Calculo la solucion al problema
    vector<int> asignation = get_best_solution(worker_matrix);

    // Muestro el coste de la solucion optima
    cout << "El coste optimo es: " << common::get_cost(worker_matrix, asignation) << endl;

    // Muestro las asignaciones hechas
```

```
    for(int i = 0; i < size; i++){
        cout << "Trabajador " << i << " asignado al trabajo " << asignation[i] << endl;
    }

    // Todo ha salido OK
    return 0;
}

// Implementacion de funciones auxiliares
//=============================================================================
vector<vector<int> > generate_permutations(int n){
    vector<vector<int> > permutations;

    // We get the base permutation to work with
    vector<int> base_perm;
    for(int i = 0; i < n; i++){
        base_perm.push_back(i);
    }

    do{
        permutations.push_back(base_perm);
    }while(next_permutation(base_perm.begin(), base_perm.end()));


    return permutations;
}

vector<int> get_best_solution(vector<vector<double> > matrix){
    // Vector que almacena las tareas que todavia no han sido asignadas
    vector<bool> available_tasks(matrix.size(), true);

    // Vector con la solucion parcial
    vector<int> asignation;

    // Asigno a cada trabajador la tarea con menor coste que quede disponible
    for(int i = 0; i < matrix.size(); i++){
        // Busco la mejor tarea que quede disponible
        int current_task = find_best_task(matrix[i], available_tasks);

        // Insertamos la tarea a la asignacion
        asignation.push_back(current_task);

        // Esa tarea ya no esta disponible
        available_tasks[current_task] = false;
    }

    return asignation;
}

int find_best_task(vector<double>tasks_cost, vector<bool> available){
    // Encuentro la primera tarea disponible para comenzar las comparaciones
    int best_task = 0;
    while(available[best_task] == false){
        best_task++;
    }
```

```
    double best_cost = tasks_cost[best_task];

    // Comparo todos los costes que quedan disponibles
    for(int i = best_task; i < tasks_cost.size(); i++){
        if(available[i] == true){ // La tarea sigue disponible
            if(tasks_cost[i] < best_cost){
                best_cost = tasks_cost[i];
                best_task = i;
            }
        }
    }

    return best_task;
}
```

worker_perturbation.cpp

Resolución usando **perturbaciones**, para ser usado partiendo de una estimación inicial.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "worker_common.hpp"
using namespace std;

// Declaracion de funciones auxiliares
//================================================================================

/**
 * @brief Calcula la asignacion parcialmente optima dada una matriz de costes,
 *        usando un algoritmo greedy a partir de una solucion parcial por insercion
 * @param matrix, la matriz que almacena los costes
 * @param num_permutations, el numero de perturbaciones que se van a realizar para mejorar la
    matriz
 * @return la asignacion parcialmente optima
 * */
vector<int> get_best_solution(vector<vector<double> > matrix, int num_permutations);

/**
 * @brief Encuentra el trabajador con peor coste
 * @param matrix, la matriz con los costes
 * @param asignation, la asignacion de la que se tiene que encontrar el peor trabajador
 * */
int find_worst_worker(vector<vector<double> > matrix, vector<int> asignation);

/**
 * @brief Encuentra cual es la mejor permutacion para mejorar el coste de una asignacion
 * @param matrix, la matriz con los costes asociados
 * @param base_asignation, asignacion base que tenemos que mejorar
 * @param worst_worker, trabajador con pero coste, el cual se tiene que intercambiar con otro
 * @return el indice del trabajador con el que se tiene que intercambiar el peor trabajador
 * */
int find_best_permutation(vector<vector<double> > matrix, vector<int> base_asignation, int
    worst_worker);

```

```
/**
 * @brief Realiza un intercambio entre trabajadores
 * @param base_asignation, asignacion sobre la que realizamos el intercambio, SE MODIFICA
 * @param first_worker, indice del trabajador que se intercambia
 * @param second_worker, indice del otro trabajador que se intercambia
 * */
void perturbate(vector<int> & base_asignation, int first_worker, int second_worker);

/**
 * @brief Calcula la asignacion parcialmente optima dada una matriz de costes,
 *        usando un algoritmo greedy
 * @param matrix, la matriz que almacena los costes
 * @return la asignacion parcialmente optima
 * */
vector<int> get_insertion_solution(vector<vector<double> > matrix);

/**
 * @brief Halla dado un vector de costo de tareas, la mejor tarea que se encuentra disponible
 * @param tasks_cost, vector con los costos de las tareas
 * @param available, vector para controlar si las tareas se encuentra o no disponibles
 * */
int find_best_task(vector<double>tasks_cost, vector<bool> available);

// Funcion principal
//=============================================================================
int main(){
    // Parametros para hacer pruebas con las entradas
    int size = 400;
    int num_permutations = 20;

    // Inicio la generacion de numeros aleatorios
    common::startRandom();

    // Tomo una matriz con los datos del problema
    vector<vector<double> > worker_matrix = common::generate_matrix(size);

    // Calculo la solucion al problema
    vector<int> asignation = get_best_solution(worker_matrix, num_permutations);

    // Muestro las asignaciones hechas
    for(int i = 0; i < size; i++){
        cout << "Trabajador " << i << " asignado al trabajo " << asignation[i] << endl;
    }

    // Muestro el coste de la solucion optima
    cout << "El coste optimo es: " << common::get_cost(worker_matrix, asignation) << endl;

    // Todo ha salido OK
    return 0;
}

// Implementacion de funciones auxiliares
//=============================================================================
vector<int> get_best_solution(vector<vector<double> > matrix, int num_permutations){
    // Parto de la solucion parcial del greedy por insercion
```

```
90      vector<int> base_asignation = get_insertion_solution(matrix.size());
91
92      // Hacemos el numero de perturbaciones dado
93      for(int i = 0; i < num_permutations; i++){
94          // Busco cual es el trabajador con mayor coste
95          int worst_worker = find_worst_worker(matrix, base_asignation);
96
97          // Busco cual es la mejor permutacion
98          int best_permutation = find_best_permutation(matrix, base_asignation, worst_worker);
99
100         // Hago el cambio si hay un cambio a mejor
101         perturbate(base_asignation, worst_worker, best_permutation);
102     }
103
104     return base_asignation;
105 }
106
107 int find_worst_worker(vector<vector<double> > matrix, vector<int> asignation){
108     double worst_cost = matrix[0][asignation[0]];
109     int worst_worker = 0;
110
111     for(int i = 0; i < asignation.size(); i++){
112         if(matrix[i][asignation[i]] > worst_cost){
113             worst_cost = matrix[i][asignation[i]];
114             worst_worker = i;
115         }
116     }
117
118     return worst_worker;
119 }
120
121 int find_best_permutation(vector<vector<double> > matrix, vector<int> base_asignation, int
        worst_worker){
122     // Coste base y coste de cada iteracion
123     double base_cost = common::get_cost(matrix, base_asignation);
124
125     // Genero una perturbacion aleatoria
126     int best_permutation = common::random_int(0, base_asignation.size() -1);
127     vector<int> current_perturbation = base_asignation;
128     perturbate(current_perturbation, worst_worker, best_permutation);
129
130     // Calculo la ganancia de la perturbacion aleatoria
131     double current_cost = common::get_cost(matrix, current_perturbation);
132     double best_gain = base_cost - current_cost;
133
134     // Busco la mejor perturbacion
135     for(int i = 0; i < base_asignation.size(); i++){
136         // Tomo la perturbacion actual
137         current_perturbation = base_asignation;
138         perturbate(current_perturbation, worst_worker, i);
139
140         // Calculo el nuevo coste de la perturbacion y la ganancia
141         current_cost = common::get_cost(matrix, current_perturbation);
142         double current_gain = base_cost - current_cost;
143
```

```
        // Hago la comparativa
        if(current_gain < best_gain){
            best_gain = current_gain;
            best_permutation = i;
        }
    }

    return best_permutation;
}

void perturbate(vector<int> & base_asignation, int first_worker, int second_worker){
    int tmp = base_asignation[first_worker];
    base_asignation[first_worker] = base_asignation[second_worker];
    base_asignation[second_worker] = tmp;
}

vector<int> get_insertion_solution(vector<vector<double> > matrix){
    // Vector que almacena las tareas que todavia no han sido asignadas
    vector<bool> available_tasks(matrix.size(), true);

    // Vector con la solucion parcial
    vector<int> asignation;

    // Asigno a cada trabajador la tarea con menor coste que quede disponible
    for(int i = 0; i < matrix.size(); i++){
        // Busco la mejor tarea que quede disponible
        int current_task = find_best_task(matrix[i], available_tasks);

        // Insertamos la tarea a la asignacion
        asignation.push_back(current_task);

        // Esa tarea ya no esta disponible
        available_tasks[current_task] = false;
    }

    return asignation;
}

int find_best_task(vector<double>tasks_cost, vector<bool> available){
    // Encuentro la primera tarea disponible para comenzar las comparaciones
    int best_task = 0;
    while(available[best_task] == false){
        best_task++;
    }
    double best_cost = tasks_cost[best_task];

    // Comparo todos los costes que quedan disponibles
    for(int i = best_task; i < tasks_cost.size(); i++){
        if(available[i] == true){ // La tarea sigue disponible
            if(tasks_cost[i] < best_cost){
                best_cost = tasks_cost[i];
                best_task = i;
            }
        }
    }
```

```
199
200     return best_task;
201 }
```

Anexo II. Tiempos

Datos 1. TSP

Cercanía		
n	Dist.	t (ms)
100	1278.99	935
115	1462.46	300
130	1781.19	410
145	1907.6	479
160	2474.64	586
175	2787.39	643
190	3249.56	766
205	3507.7	876
220	3973.02	1265
235	3983.08	1178
250	4596.83	1183
265	5095.48	1288
280	5373.6	1388
295	6362.87	1541

Inserción		
n	Dist.	t (ms)
100	1138	577206
115	1364.62	1014658
130	1618.9	1523364
145	1893.79	2311261
160	2184.95	3325929
175	2555.25	4614364
190	2924.26	6693015
205	3183.27	8416194
220	3559.8	10847461
235	4108.07	14116260
250	4459.9	17317916
265	4645.52	22675340
280	5025.57	27542775
295	5353.33	33257464

Perturbaciones		
n	Dist.	t (ms)
10	35	9
30	223	40
50	409	91
70	706	140
90	907	156
110	1355	220
130	1691	300
150	2234	377
170	2690	477
190	2991	590

Datos 2. *Worker*

Inserción		
n	**Coste**	**t** (ms)
100	450	597
300	1646	3346
500	3233	8672
700	4627	16544
900	5786	25110
1100	8576	40257
1300	9889	54961
1500	11342	74864
1700	15010	98253
1900	13022	116179
2100	13659	144524
2300	18817	167708
2500	17637	190472
2700	20622	231699
2900	20158	260600
3100	25547	304797
3300	23987	347594
3500	27613	389803
3700	27777	432598
3900	29223	491516

Perturbaciones		
n	**Coste**	**t** (ms)
1	0	669
11	74	7924
21	353	24913
31	659	51275
41	1255	85161
51	2059	131537
61	2604	183380
71	3925	248788
81	5359	319110
91	6575	405153