UNIVERSIDAD
DE GRANADA

Algoritmos Greedy (o voraces)

Algorítmica. Práctica 3

Jose Alberto Hoces Castro
Javier Gómez López
Manuel Moya Martín Castaño

Mayo 2022

Contenidos

Objetivo de la práctica

Aprender a analizar un problema y resolverlo mediante la técnica Greedy, además de justificar su utilidad para resolver problemas de forma muy eficiente, obteniendo la solución óptima o muy cercana a la óptima.

Ejercicio 1. Contenedores

Enunciado

Se tiene un buque mercante cuya capacidad de carga es de K *toneladas y un conjunto de contenedores* $c_1, \ldots, c_n$ *cuyos pesos respectivos son* $p_1, \ldots, p_n$ *(expresados también en toneladas). Teniendo en cuenta que la capacidad del buque es menor que la suma total de los pesos de los contenedores:*

Primer ejercicio

Diseñe un algoritmo que maximice el número de contenedores cargados, y demuestre su optimalidad.

Primer ejercicio. Planteamiento del algoritmo

- Como queremos cargar el máximo número de contenedores, empezaremos cargando los más **pequeños**.
- Ordenamos de **menor a mayor** peso los contenedores.
- Empezamos a cargar los de menor peso hasta que superemos las K toneladas del buque mercante.
- Todo esto lo simulamos con un vector de enteros en nuestro código, el cual tenemos a continuación.

Primer ejercicio. Código

```
1  int contenedoresGreedy1(int *T, int n){
2
3          int used = 0;
4          int result = 0;
5      vector<int> myvector(T,T+n);
6      sort(myvector.begin(),myvector.end());
7
8      for(int i = 0; (i < n) && (used <= n); i++){
9          used += T[i];
10         result++;
11     }
12
13     return result;
14 }
```

Primer ejercicio. Enfoque Greedy

Las 6 características de nuestro problema que hacen que lo identifiquemos como problema Greedy son:

- **Un conjunto de candidatos**: En este caso, los contenedores a cargar.
- **Una lista de candidatos ya usados**: Los contenedores que ya han sido cargados.
- **Un criterio que dice cuándo un conjunto de candidatos forma una solución**: El criterio es que la suma de los pesos de un conjunto de contenedores no sea superior a las K toneladas del buque.

Primer ejercicio. Enfoque Greedy

- **Un criterio que dice cuándo un conjunto de candidatos es factible (podrá llegar a ser una solución)**: el conjunto de contenedores que se evalúe no debe superar en peso las K toneladas del buque.
- **Una función de selección que indica en cualquier instante cuál es el candidato más prometedor de los no usados todavía**: El contenedor de menor peso de los que aún no están cargados, de ahí que los ordenemos de menor a mayor peso.
- **La función objetivo que intentamos optimizar**: El número de contenedores a cargar, es lo que queremos maximizar.

Primer ejercicio. Estudio de la optimalidad

Sea $T = \{c_1, ..., c_n\}$ y llamemos $S = \{c_1, ..., c_m\}$ a la solución de nuestro algoritmo Greedy.

$$\sum_{c_i \in S} p_i = \sum_{i=1}^{m} p_i \leq K \text{ y } \sum_{i=1}^{m+1} p_i > K$$

Sea $U \subset T$ con un número mayor de contenedores que S y veamos que no es solución, es decir, que $\sum_{c_i \in U} p_i > K$.

Primer ejercicio. Estudio de la optimalidad

$$\sum_{c_i \in U} p_i = \sum_{c_i \in S \cap U} p_i + \sum_{c_i \in U \setminus S} p_i$$

$$\downarrow$$

$$\sum_{c_i \in S \cap U} p_i + \sum_{c_i \in U \setminus S} p_i = \sum_{c_i \in S \cap U} p_i + \sum_{c_i \in R} p_i + \sum_{c_i \in U \setminus S \setminus R} p_i$$

$$\downarrow$$

$$\sum_{c_i \in S \cap U} p_i + \sum_{c_i \in R} p_i + \sum_{c_i \in U \setminus S \setminus R} p_i \geq \sum_{c_i \in S \cap U} p_i + \sum_{c_i \in S \setminus U} p_i + \sum_{c_i \in U \setminus S \setminus R} p_i$$

$$\downarrow$$

$$\sum_{c_i \in S \cap U} p_i + \sum_{c_i \in S \setminus U} p_i + \sum_{c_i \in U \setminus S \setminus R} p_i = \sum_{c_i \in S} p_i + \sum_{c_i \in U \setminus S \setminus R} p_i > K$$

Luego hemos demostrado que $\sum_{c_i \in U} p_i > K$ y por lo tanto, U no es solución.

Segundo ejercicio

Diseñe un algoritmo que intente maximizar el número de toneladas cargadas.

Segundo ejercicio. Planteamiento del algoritmo

- Como queremos cargar el máximo número de toneladas, empezaremos cargando los más **pesados**.
- Ordenamos de **mayor a menor** peso los contenedores.
- Empezamos a cargar los de mayor peso hasta que superemos las K toneladas del buque mercante.
- Todo esto lo simulamos con un vector de enteros en nuestro código, el cual tenemos a continuación.

Segundo ejercicio. Código

```
1  int contenedoresGreedy2(int *T, int n){
2
3          int used = 0;
4      vector<int> myvector(T,T+n);
5      sort(myvector.begin(),myvector.end(), greater<int>());
6
7      for(int i = 0; (i < n) && (used <= n); i++){
8          used += T[i];
9      }
10
11     return used;
12 }
```

Segundo ejercicio. Estudio de la optimalidad

[5, 4, 6, 1, 1, 2, 7, 9, 8, 3] K = 10

↓

[9, 8, 7, 6, 5, 4, 3, 2, 1, 1]

Solución aportada por nuestro algoritmo: [9]

Solución óptima: [1,2,3,4]

Ejercicio 2. El problema del viajante de comercio

Enunciado

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.

TSP. Heurística del vecino más cercano

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 cubrir todos los nodos.

Heurística del vecino más cercano. Código

```
def get_best_solution(points):
    road = []
    order = []

    distance_matrix = gen_distance_matrix(points, len(points))

    #We start always at first point
    last_point = 0
    road.append(points[last_point])
    order.append(0)

    while len(road) < len(points):
        best_position = get_min_row_element(distance_matrix,
    last_point)

        road.append(points[best_position])
        order.append(best_position)

        clean_position(distance_matrix, last_point)

        last_point = best_position

    road_distance = get_road_distance(road)

    return road, road_distance, order
```

Heurística del vecino más cercano. Análisis Teórico

$$T(n) \in O(n^2)$$

Heurística del vecino más cercano. Análisis empírico

Heurística del Vecino más cercano	
Ciudades (n)	**Tiempo (s)**
358	0.134928381001373
537	0.301542887000323
716	0.571918544999789
895	0.886006714001269
1074	1.24296611099999
1253	1.64954555899931
1432	2.64052301800075
1611	2.85592838900084
1790	3.59707787300067
1969	4.38948754500052
2148	5.39098084799844
2327	6.98254896399885
2506	7.10732670200014
2685	8.94702127299934
2874	9.27537115900122
3053	9.69826381100029
3232	11.1923151139999
3411	12.4046790310003
3590	12.853421451
3769	14.3906136509995
3948	15.625681644
4127	17.1084850459993
4306	18.5073586279996
4461	19.8675596370013

Tabla 1: Experiencia empírica de el vecino más cercano

Heurística del vecino más cercano. Análisis Híbrido

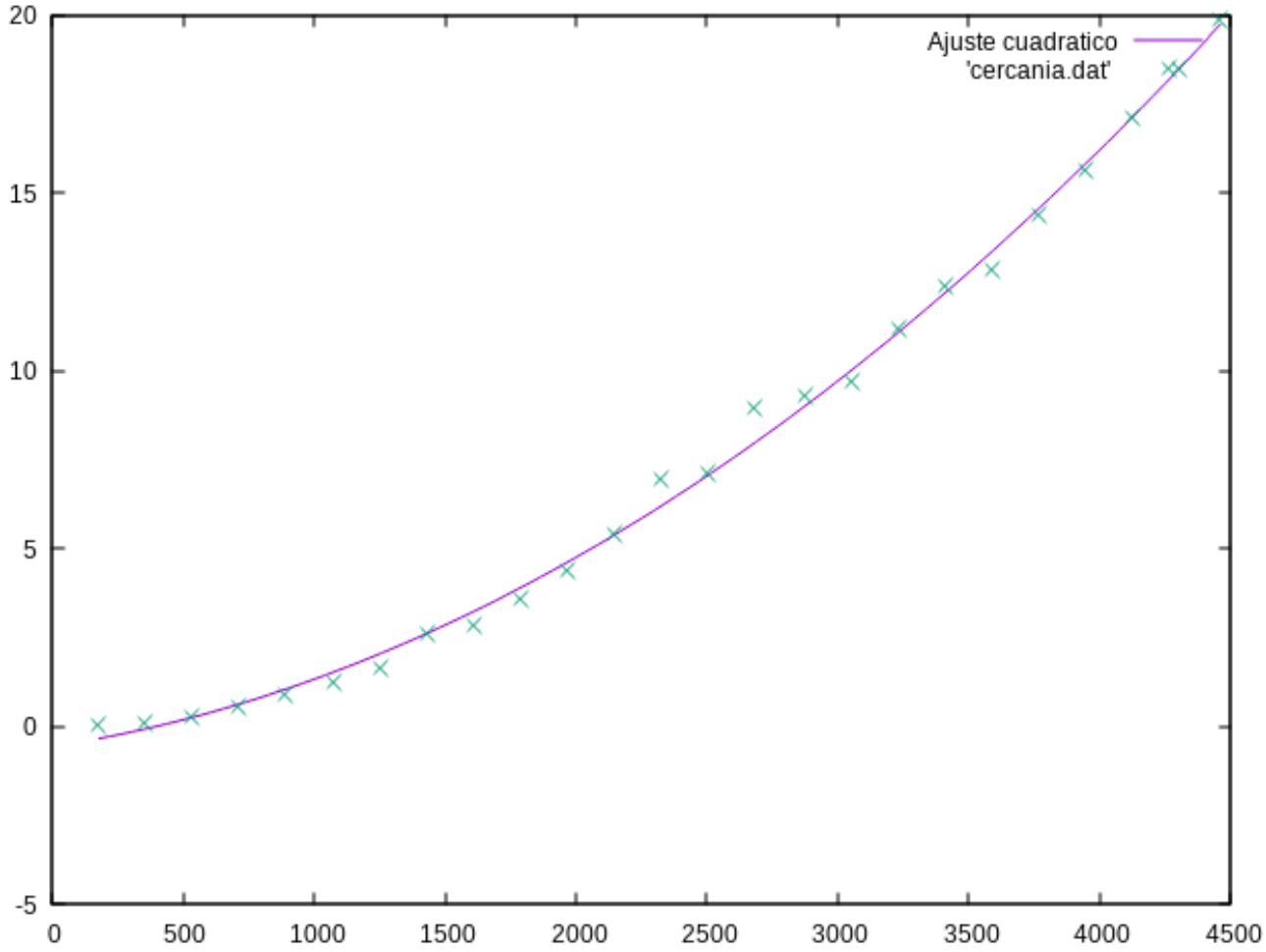


Figura 1: Gráfica con los tiempos de ejecución del vecino más cercano.
R^2 = 0.869828

Heurística del vecino más cercano. Resultados

- `ulysses16.tsp`:

$$[0, 7, 15, 12, 11, 13, 6, 5, 14, 4, 8, 9, 3, 1, 2, 10] \qquad D = 103$$

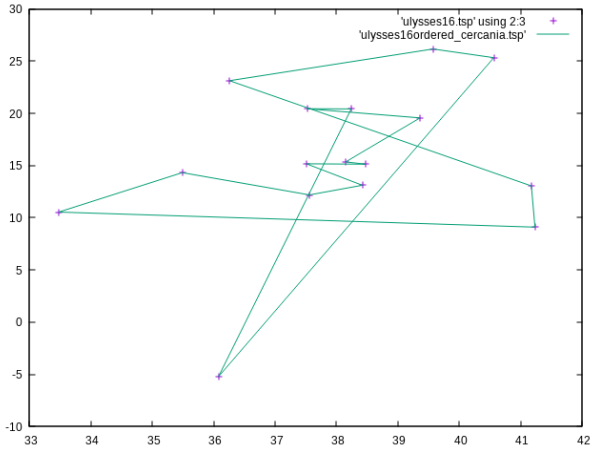
- `bayg29.tsp`:

$$[0, 27, 5, 11, 8, 4, 20, 1, 19, 9, 3, 14, 17, 13, 21, 16, 10, 18, 24,$$
$$6, 22, 26, 7, 23, 15, 12, 28, 25, 2] \qquad D = 10209$$

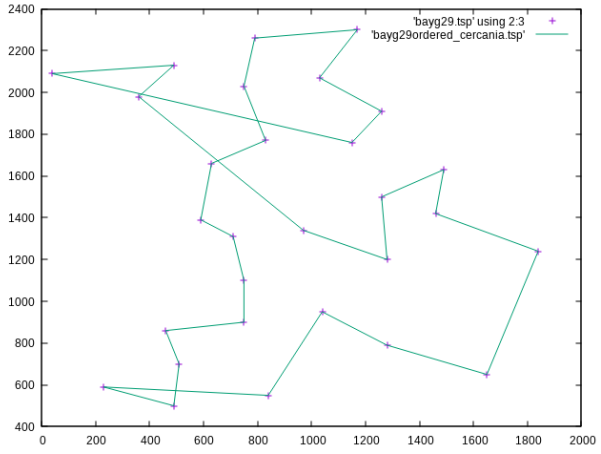
- `eil76.tsp`:

$$[0, 72, 32, 62, 15, 2, 43, 31, 8, 38, 71, 57, 9, 37, 64, 10, 65, 52, 13, 18, 34, 6, 7, 45,$$
$$33, 51, 26, 44, 28, 47, 46, 20, 73, 27, 61, 1, 29, 3, 74, 75,$$
$$66, 25, 11, 39, 15, 50, 5, 67, 4, 36, 19, 69, 59, 70, 35, 68, 60, 2, 1,$$
$$41, 40, 42, 22, 55, 48, 23, 17, 49, 24, 54, 30, 58, 53, 12, 56, 14, 63] \qquad D = 642$$

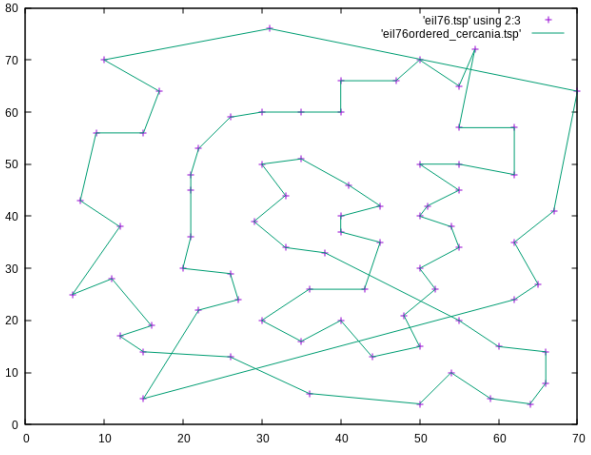
Heurística del vecino más cercano. Gráficos



Ulysses



Bayg



Eil

TSP. Heurística de inserción

Veamos las 6 características de nuestro problema Greedy:

- **Un conjunto de candidatos**: En este caso, las ciudades por las que pasar.
- **Una lista de candidatos ya usados**: Los ciudades por las que ya se ha pasado. Cabe resaltar que se debe comenzar por un recorrido inicial de 3 ciudades previamente. Para escoger estas ciudades simplemente lo haremos escogiendo la ciudad más al oeste, la más al este y la más al norte.
- **Un criterio que dice cuándo un conjunto de candidatos forma una solución**: El criterio es que el recorrido que se haga forme un circuito pasando por todas las ciudades una sola vez.
- **Un criterio que dice cuándo un conjunto de candidatos es factible (podrá llegar a ser una solución)**: En caso de que no se repita ningún nodo (ciudad), el conjunto de candidatos es factible.

TSP. Heurística de inserción

- **Una función de selección que indica en cualquier instante cuál es el candidato más prometedor de los no usados todavía**: Utilizaremos un criterio que denominaremos inserción mas económica: de entre todas las ciudades no visitadas, elegimos aquella que provoque el menor incremento en la longitud total del circuito. En otras palabras, cada ciudad debemos insertarla en cada una de las soluciones posibles y quedarnos con la ciudad (y posición) que nos permita obtener un circuito de menor longitud. Seleccionaremos aquella ciudad que nos proporcione el mínimo de los mínimos calculados para cada una de las ciudades
- **La función objetivo que intentamos optimizar**: El coste del recorrido total del circuito.

Heurística de inserción. Código

```
ALGORITMO INSERCION TSP (G=(V,R))
C = V [C = Lista de Candidatos]
S = Vacio [S = Conjunto Solucion]
Crear a partir de R recorridoInicial
S << {este, oeste, norte}
C = C\{este, oeste, norte}
Fin de Crear
Repetir hasta que cardinal(C)= Vacio
    Crear vector pCandidatos
    Para cada s en S
        Para cada c en C
            Calcular distancia(s,c)
            Seleccionar punto cercano de C al punto "s"
        Fin-Para
    Almacenar (c,distancia) >> puntosCandidatos
    Fin de Para
    Seleccionar p en puntosCandidatos que produzca menor
    incremento
    Insertar p en T
    C = C\{p}
Fin de Repetir
```

Heurística de inserción. Análisis Teórico

$$T(n) \in O(n^3)$$

Heurística de inserción. Análisis empírico

Heurística de inserción	
Ciudades (n)	**Tiempo (s)**
358	0.276428
537	0.925309
716	2.23608
895	4.21995
1074	7.40272
1253	0.0587605
1432	0.275992
1611	0.943963
1790	2.39547
1969	4.42937
2148	7.34342
2327	11.8085
2506	17.2049
2685	24.4085
2874	33.8853
3053	44.5835
3232	57.4245
3411	73.2457
3590	92.1468
3769	113.815
3948	139.366
4127	165.838
4306	200.112
4461	235.333

Tabla 2: Experiencia empírica de inserción

Heurística de inserción. Análisis Híbrido

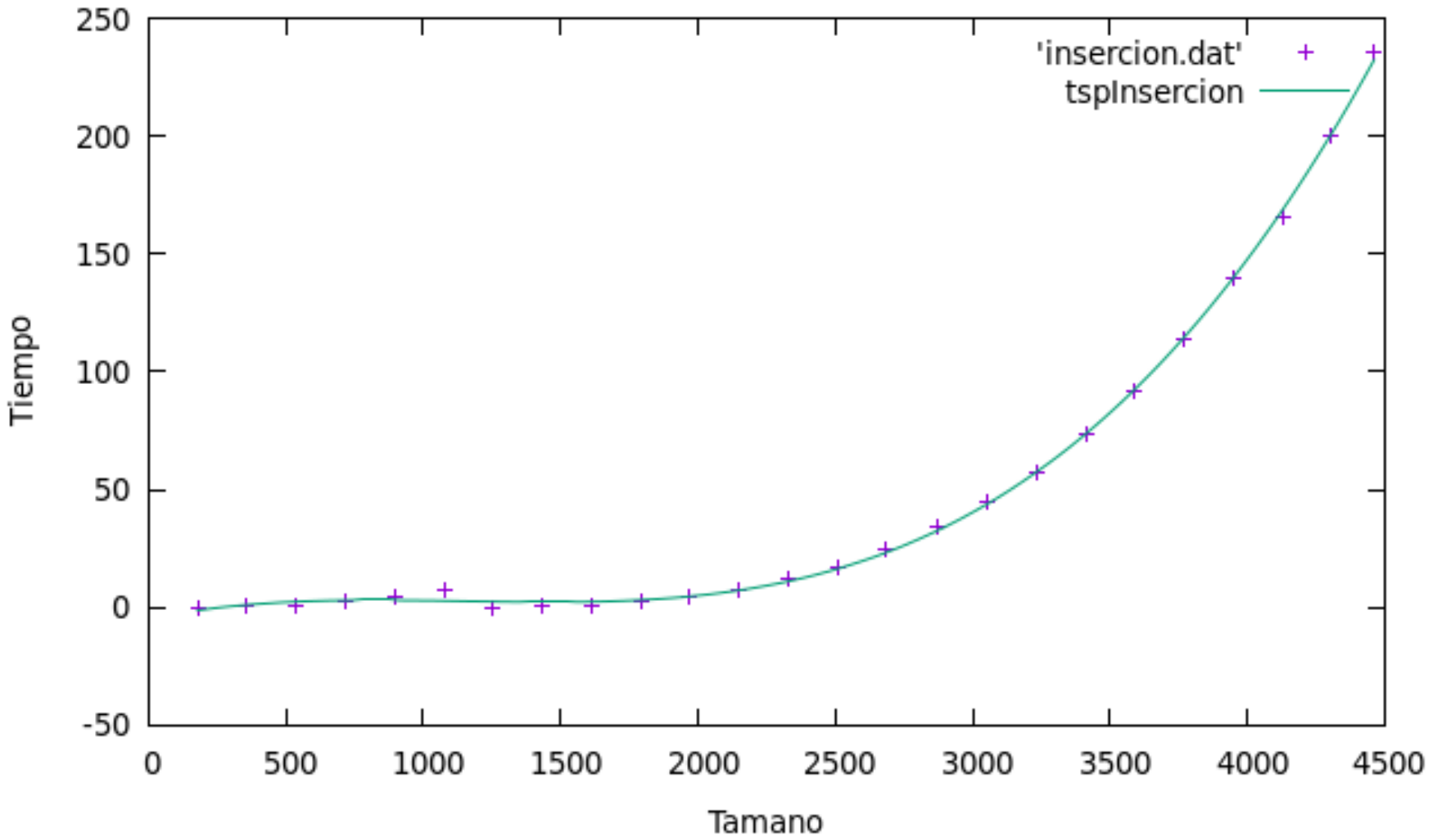


Figura 2: Gráfica con los tiempos de ejecución de insercion. $R^2 = 0.85647$

Heurística de inserción. Resultados

- `ulysses16.tsp`:

  $[5, 6, 7, 10, 12, 14, 15, 13, 16, 1, 4, 8, 9, 11, 2, 3]$ $D = 99$

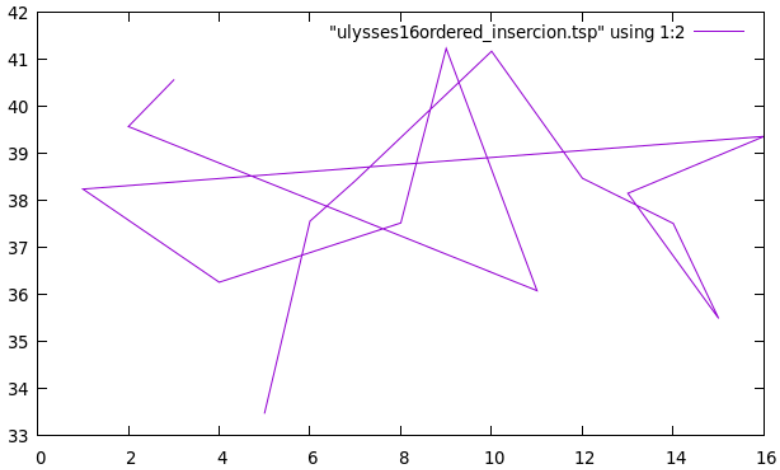
- `bayg29.tsp`:

  $[3, 23, 12, 6, 5, 26, 29, 21, 2, 20, 10, 13, 4, 15, 19, 25, 7, 18,$
  $14, 22, 11, 17, 9, 28, 1, 24, 27, 16, 8]$ $D = 10048$

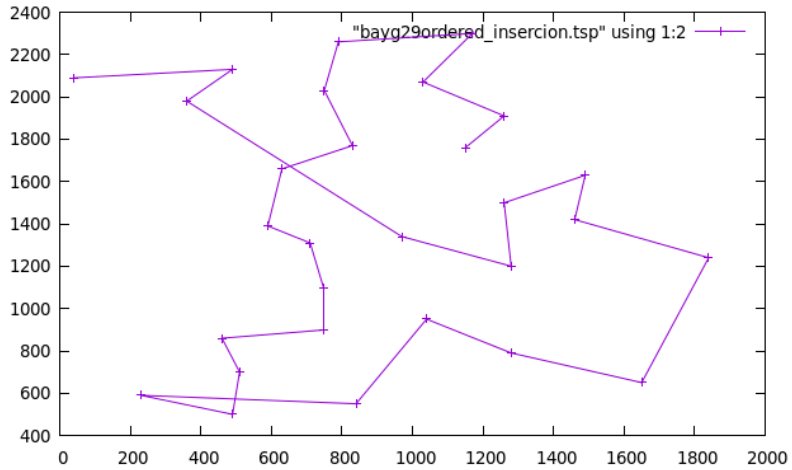
- `eil76.tsp`:

  $[56, 23, 63, 33, 73, 62, 22, 28, 61, 74, 30, 48, 5, 15, 57, 37, 20, 70, 60,$
  $71, 47, 36, 69, 21, 29, 45, 27, 13, 54, 52, 34, 67, 26, 76, 75, 4, 68, 6, 51,$
  $17, 12, 40, 46, 8, 35, 53, 11, 66, 65, 38, 10, 58, 72, 39, 9, 32, 50, 25, 55,$
  $18, 44, 3, 14, 19, 7, 2, 1, 43, 41, 42, 64, 16, 49, 24, 59, 31]$ $D = 611$

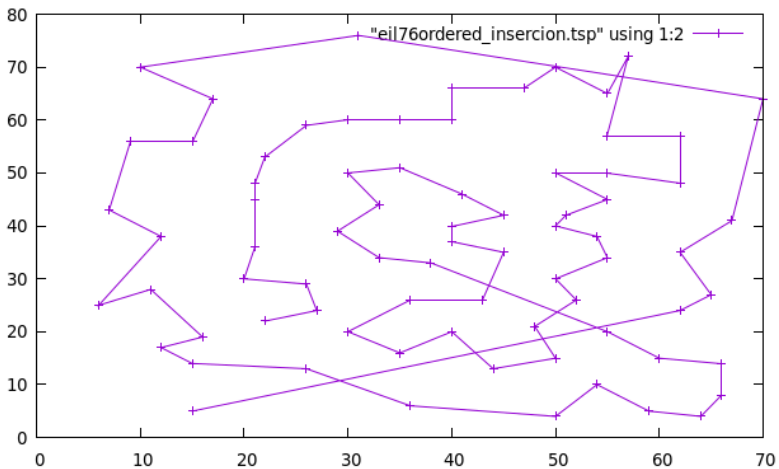
Heurística de insercion. Gráficos



Ulysses



Bayg



TSP. Heurística de perturbaciones

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

Heurística de perturbaciones. Código

```
def perturbate(road, orden, pos):
    current_perb = road
    best_gain = math.inf
    best_perturbation = pos
    base_distance = get_road_distance(road)

    for i in range(len(road)):
        current_perb = get_swap(road, pos, i)
        swap_distance = get_road_distance(current_perb)

        if swap_distance < base_distance:
            best_perturbation = i
            base_distance = swap_distance

    road = get_swap(road, pos, best_perturbation)
    orden = get_swap(orden, pos, best_perturbation)

def get_best_solution_perturbations(points, orden, perturbations
    ):
    base_road = points

    for i in range(perturbations):
        pos = get_worst_node(points)

        perturbate(base_road,orden, pos)

    return base_road, get_road_distance(base_road), orden
```

Heurística de perturbaciones. Análisis teórico

$$T(n) \in O(n^2 \cdot perturbaciones) \Rightarrow T(n) \in O(n^2)$$

Heurística de perturbaciones. Análisis empírico

Heurística de perturabciones	
Ciudades (n)	**Tiempo (s)**
358	7.09205518299859
537	14.8038363970009
716	27.0365697790003
895	42.5021277929991
1074	60.6207957549996
1253	79.9574952149997
1432	106.705971191001
1611	133.111995577001
1790	163.618691645999
1969	198.794982856998
2148	241.148463359001
2327	275.541576210999
2506	323.217905321999
2685	371.290206549998
2874	429.912281959001
3053	485.322751331001
3232	545.820733338998
3411	603.358232573999
3590	678.706273265001
3769	751.554062298001
3948	825.585648235003
4127	890.451960293001
4306	973.566447267
4461	1045.84781561849

Tabla 3: Experiencia empírica de perturbaciones

Heurística de perturbaciones. Análisis híbrido

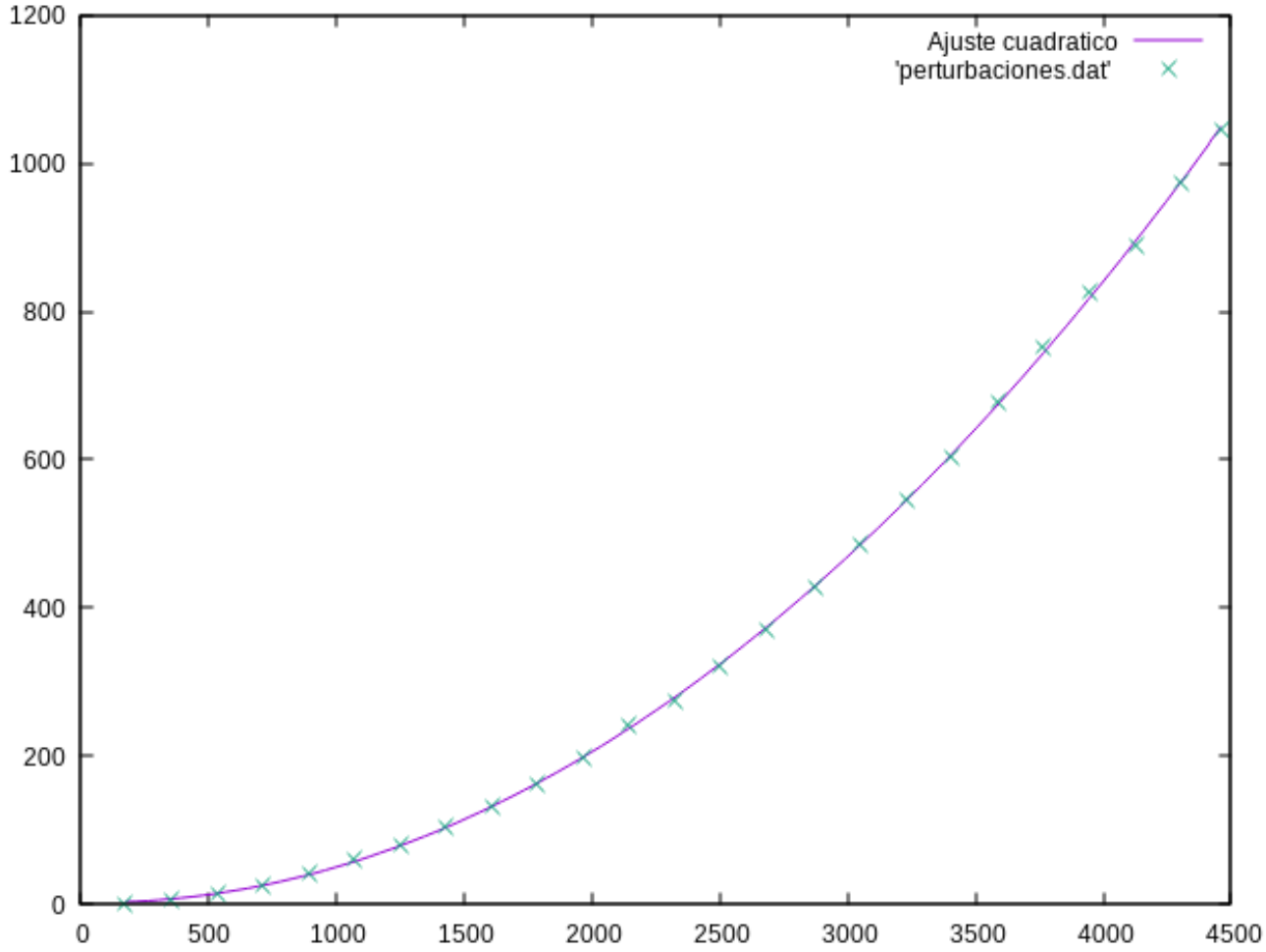


Figura 3: Gráfica con los tiempos de ejecución del vecino más cercano.
R^2 = 0.93507

Heurística de perturbaciones. Resultados

- `ulysses16.tsp`:

$$[0, 7, 15, 12, 11, 13, 6, 5, 14, 4, 8, 9, 2, 1, 3, 10] \qquad D = 101$$

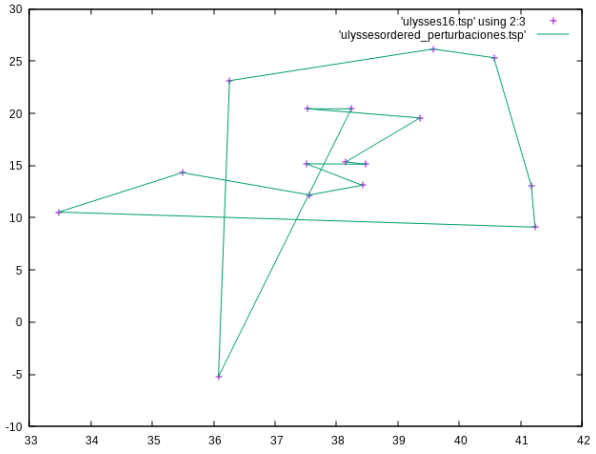
- `bayg29.tsp`:

$$[0, 27, 5, 11, 8, 4, 20, 1, 19, 9, 3, 14, 17, 13, 21, 16, 10, 18, 24, 6, 22, 26, 7,$$
$$23, 15, 12, 28, 25, 2] \qquad D = 10209$$

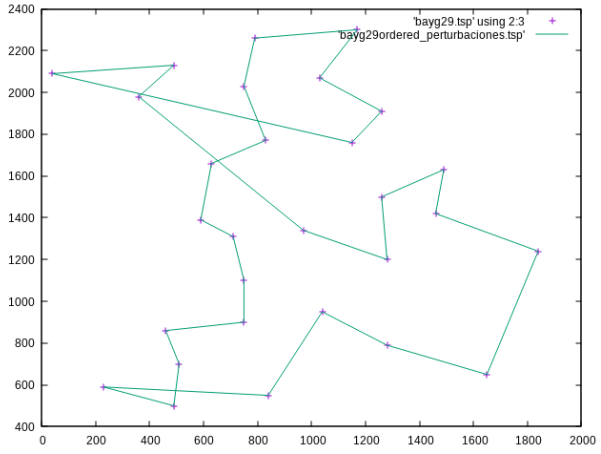
- `eil76.tsp`: Aplicando 10 perturbaciones, obtenemos que el mejor orden (teniendo en cuenta el orden del fichero original):

$$[0, 72, 32, 62, 15, 2, 43, 31, 8, 38, 71, 57, 9, 37, 64, 10, 65, 52, 13, 18, 34, 6, 7, 45, 33, 51, 26, 44, 28,$$
$$47, 46, 20, 73, 27, 61, 1, 29, 3, 74, 75, 66, 25, 11, 39, 16, 50, 5, 67, 4, 36, 19, 69, 59, 70, 35, 68, 60, 2,$$
$$1, 41, 40, 42, 22, 55, 48, 23, 17, 49, 24, 54, 30, 58, 53, 12, 56, 14, 63] \qquad D = 642$$

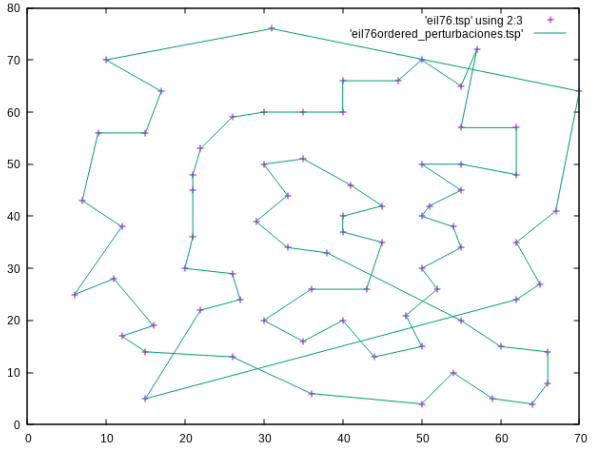
Heurística de perturbaciones. Gráficos



Ulysses



Bayg



Eil

Comparación de las distinas heurísticas

Los tres algoritmos que tenemos son: cercanía (el vecino más cercano), inserción y por perturbaciones. Veremos la gráfica comparativa de los tiempos de cada heurística.

Comparación de las distinas heurísticas

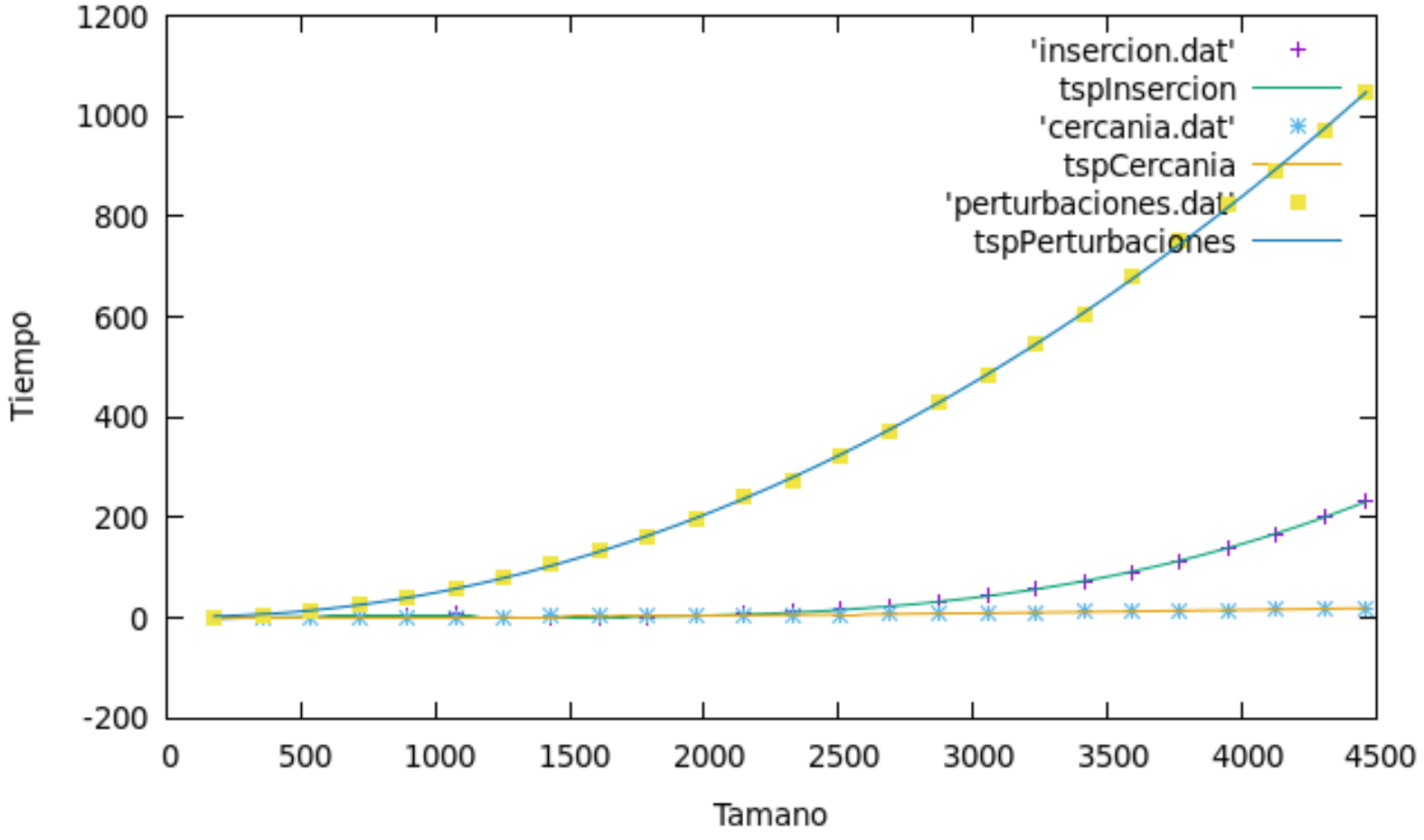


Figura 4: Gráfica comparativa con los tiempos de ejecución de las distintas heurísticas

Comparación de las distinas heurísticas

Heurística de cercania	
Ciudades (n)	**Distancia ()**
358	22268
537	28232
716	38482
895	45794
1074	54590
1253	64034
1432	72686
1611	81277
1790	92277
1969	97931
2148	106301
2327	118281
2506	127627
2685	132398
2874	149824
3053	154575
3232	167010
3411	173936
3590	181619
3769	193185
3948	199550
4127	208015
4306	222743
4461	229963

Heurística de insercion	
Ciudades (n)	**Distancia ()**
358	22081
537	27956
716	37891
895	45087
1074	53629
1253	63147
1432	71017
1611	80478
1790	91078
1969	97106
2148	105102
2327	108140
2506	124987
2685	130156
2874	147103
3053	152685
3232	165432
3411	170914
3590	179974
3769	190034
3948	196996
4127	205108
4306	219841
4461	226683

Heurística de perturbaciones	
Ciudades (n)	**Distancia ()**
358	22247
537	28232
716	38409
895	45789
1074	54287
1253	63952
1432	72662
1611	81077
1790	92277
1969	97913
2148	106280
2327	118281
2506	126984
2685	132239
2874	149530
3053	154265
3232	167010
3411	173936
3590	181411
3769	192804
3948	199525
4127	207947
4306	222743
4461	229960

Tabla 4: Experiencia empírica de las distintas heurísticas

Comparación de las distinas heurísticas

Vemos como los tiempos de ejecución son mayores para inserción debido a que su eficiencia es cúbica respecto a los demás, donde también se observa que la heurística de perturbaciones tiene un tiempo algo mayor que la de cercanía. Sin embargo este mayor tiempo se puede justificar a la hora de escoger estas heurísticas debido a que el algoritmo que proporciona mejores soluciones es, como se puede comprobar con los costes previamente reflejados, el que utiliza la heurística de inserción y posteriormente al que utiliza la heurística de perturbaciones. Por tanto la conclusión en este caso es que a mayor tiempo de ejecución mejor solución obtenemos y por tanto debemos de tener esto en cuenta a la hora de elegir una heurística u otra.

Conclusiones

Conclusiones

- Los algoritmos Greedy nos ahorran la elevada complejidad de los algoritmos de fuerza bruta.
- Los algoritmos más eficientes no siempre son los más útiles.
- La anterior conclusión nos lleva a que debemos saber encontrar un término medio entre eficiencia y búsqueda de la solución óptima.
- Nos interesa una buena eficiencia y una solución próxima a la óptima
- Los algoritmos Greedy no siempre nos dan la mejor solución.