UNIVERSIDAD DE GRANADA

Algoritmos Greedy (o voraces)

Algorítmica. Práctica 3

Jose Alberto Hoces Castro
Javier Gómez López
Manuel Moya Martín Castaño

Mayo 2022

Contenidos

1. Ejercicio 1. Contenedores

2. Ejercicio 2. El problema del viajante de comercio

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

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. Resultados

- `ulysses16.tsp`:

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

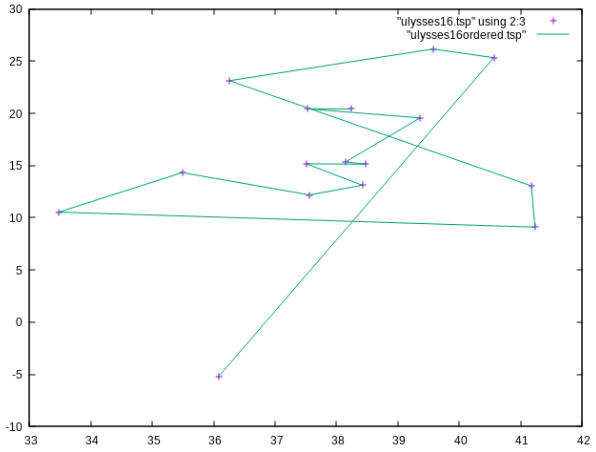
- `bayg29.tsp`:

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

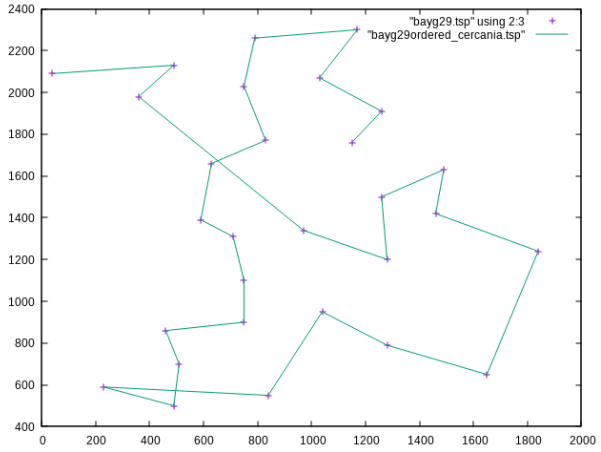
- `eil76.tsp`:

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

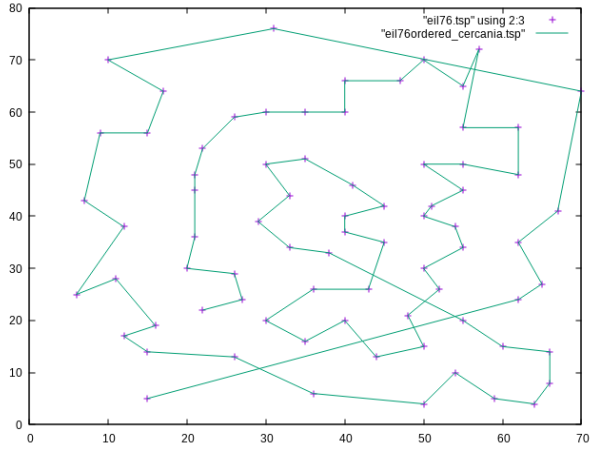
Heurística del vecino más cercano. Gráficos



Ulysses



Bayg



Eil

TSP. Heurística de inserción

Empezar insercion aqui

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

```python
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. Resultados

- `ulysses16.tsp`:

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

- `bayg29.tsp`:

$$[0, 27, 5, 11, 8, 4, 20, 1, 19, 9, 3, 14, 17, 13, 21, 16, 10, 18, 24, 6, 22, 26, 7,$$

$$23, 15, 12, 28, 25, 2] \qquad D = 10209$$

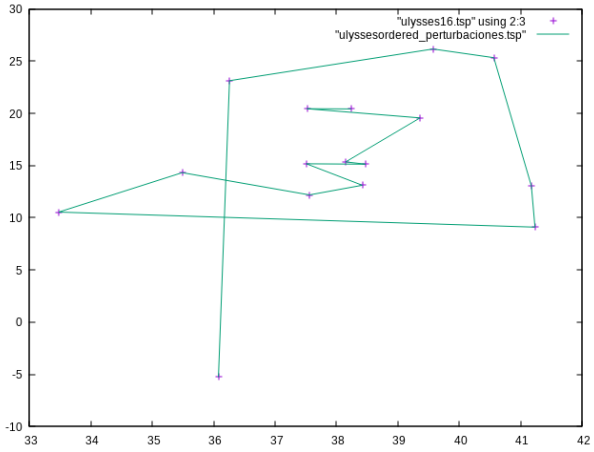
- `eil76.tsp`: Aplicando 10 perturbaciones, obtenemos que el mejor orden (teniendo en cuenta el orden del fichero original):

$$[0, 72, 32, 62, 15, 2, 43, 31, 8, 38, 71, 57, 9, 37, 64, 10, 65, 52, 13, 18, 34, 6, 7, 45, 33, 51, 26, 44, 28,$$

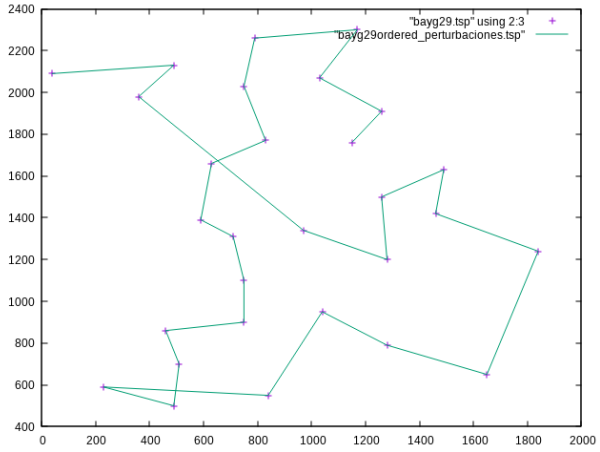
$$47, 46, 20, 73, 27, 61, 1, 29, 3, 74, 75, 66, 25, 11, 39, 16, 50, 5, 67, 4, 36, 19, 69, 59, 70, 35, 68, 60, 2,$$

$$1, 41, 40, 42, 22, 55, 48, 23, 17, 49, 24, 54, 30, 58, 53, 12, 56, 14, 63] \qquad D = 642$$

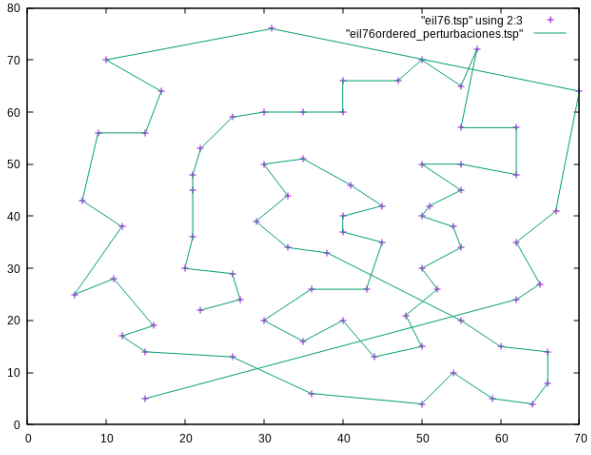
Heurística de perturbaciones. Gráficos



Ulysses



Bayg



Eil

Comparación de las distinas heurísticas

a