Práctica 4.2 -
Backtracking y Branch And Bound
Problema del Viajante del Comercio.

Adrián Portillo Sánchez
Jose Juán Pérez González
June 2015

Índice

1. Introduction

En esta práctica resolveremos el problema del viajante de comercio, el cual ya trabajamos en la práctica 3, en el cual se tiene un grafo de ciudades representadas como nodos en un mapa, y con el que se pretende encontrar un camino que recorra todas las ciudades del grafo sin repetir ninguna ciudad en el camino.
En esta ocasión resolveremos el problema con Backtracking y con Branch & Bound, por lo cual la solución que encontrará será la óptima, aunque los tamaños de datos a utilizar no podrán ser demasiado grandes, pues la carga de trabajo es muchísimo mayor en esta clase de algoritmos.

Dada una lista S de n ciudades, representadas como puntos en el plano:

$$S = [(x_0, y_0), (x_1, y_1), \dots (x_{n-1}, y_{n-1})] \subset \mathbb{R}^2 \tag{1}$$

Y definiendo la longitud de recorrer una lista como la suma de las distancias de cada ciudad a la siguiente:

$$long(S) = \sum_{i \in \mathbb{Z}_n} dist((x_i, y_i), (x_{i+1}, y_{i+1})) = \sum_{i \in \mathbb{Z}_n} \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} \tag{2}$$

El objetivo es encontrar una permutación de la lista de ciudades tal que su longitud sea mínima:

$$long(S') = long([(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]) \tag{3}$$

2. Backtracking

Utilizaremos un vector de enteros para representar una solución, donde almacenaremos los índices de las ciudades que se van recorriendo en la ruta. Como fijamos la primera ciudad y únicamente podemos pasar una vez por cada una, el espacio de soluciones tendrá un tamaño de $(n-1)!$.
La poda se realiza generando únicamente vectores que representen posibles soluciones, esto es, solo se consideran permutaciones del vector inicial $(1,\ldots,n)$. De esta forma descartamos el resto de posibles combinaciones de los elementos. Diremos que, para un nodo cualquiera, la función de poda da falso si se repite alguno de los índices.
Para aplicar backtracking al problema del TSP, utilizamos un algoritmo recursivo, el cual recibe la lista de ciudades del problema, la ruta con las ciudades que llevamos hasta el momento, su coste, y el índice de la última ciudad añadida. Calcularemos todas las ramificaciones posibles de la ruta actual, y tomaremos la mejor de estas.

```
def tsp(ciudades)
    def recorrer(indice)
        if (indice == #ciudades)
            if (coste(ruta) < mejor_coste)
                mejor_coste = coste(ruta)
                mejor_ruta  = ruta
        else
            for i in [indice..#ciudades]
                swap(ruta[i], ruta[indice])
                recorrer(indice+1)
                swap(ruta[indice], ruta[i])
    end

    mejor_coste = ∞
    ruta = [1..#ciudades]
    recorrer(1)

    return mejor_coste
end

def coste(ruta)
    return ∑ dist(ciudades[ruta[i]], ciudades[ruta[i + 1]])
           i∈ℤn
end
```

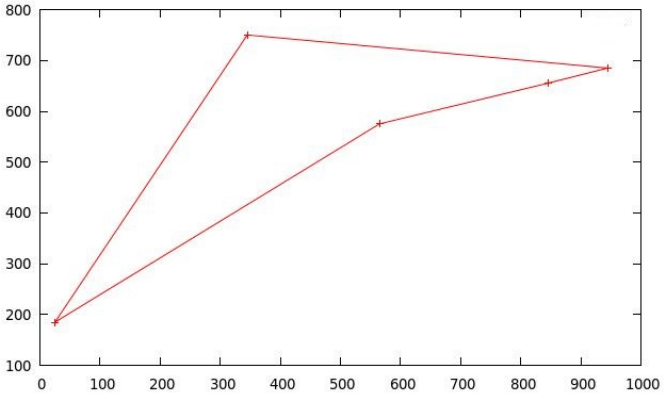
3. Branch & Bound

De nuevo, representaremos una solución como un vector de enteros de tamaño n, obteniendo el mismo espacio de soluciones de tamaño $(n-1)!$.
Dado un nodo intermedio cualquiera, los posibles hijos serán todos los otros nodos a ese hijo cualquiera.
La cota que se utiliza para la poda es el cálculo del coste (la suma de distancias entre ciudades) hasta el momento. Este coste será una cota mínima del coste de cualquier posible solución que se obtenga del nodo actual.
La poda se realiza en el momento en que la cota mínima supera al coste de la mejor solución hallada hasta el momento.
La estrategia de ramificación utilizada en este caso será Least-Cost FIFO, ya que buscamos explorar las soluciones ordenadas por el menor coste.

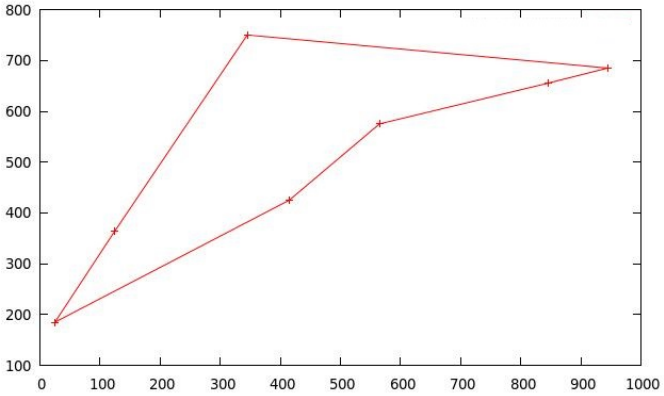
```
def tsp(ciudades)
    def recorrer()
        mejor = inicial
        actual = posibles.pop

        if (indice == #ciudades)
            if (coste(actual) < coste(mejor))
                mejor = actual

        else if (cota_inferior(actual) < coste(mejor))
            for i in [actual.indice ... #ciudades]
                hija = actual
                swap(hija[i], hija[hija.indice])
                hija.indice++
                posibles.push(hija)

        return mejor
    end

    posibles = new Priority Queue
    inicial = [1 ... #ciudades]
    inicial.indice = 1
    posibles.push(inicial)
```

```
    return recorrer()

    def cota_inferior(ruta)
        return coste + ∑_{i=1}^{#ciudades} mín_{j=1}^{#ciudades} dist(i,j)
    end

    def coste(ruta)
        return ∑_{i∈ℤ_n} dist(ciudades[ruta[i]], ciudades[ruta[i+1]])
    end
end
```

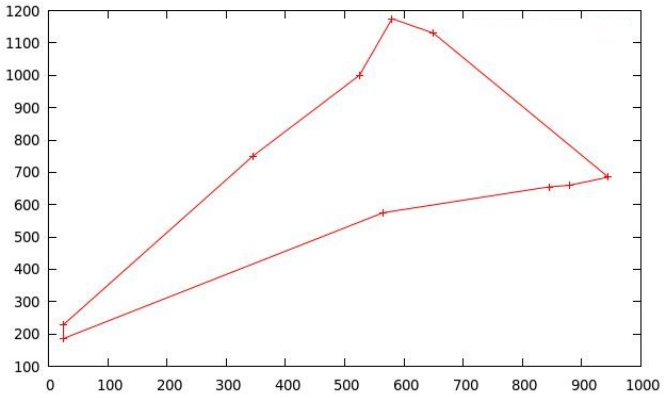
4. Resultados de la ejecución.



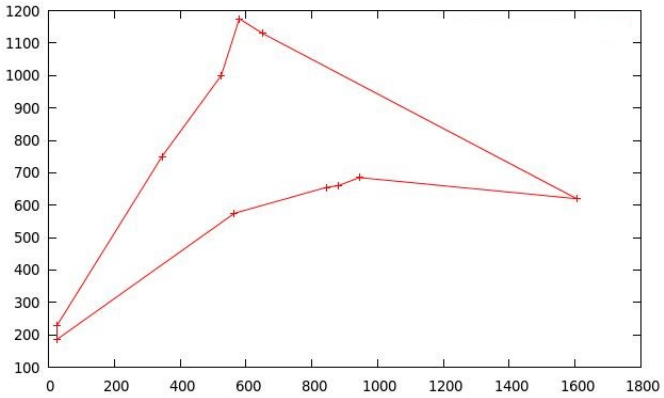
Distancia: 2314



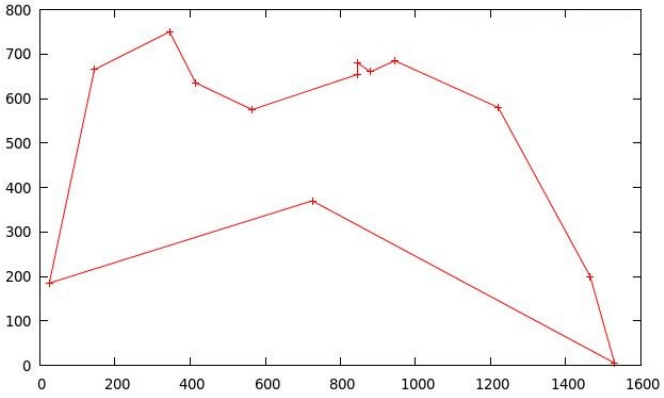
Distancia: 2318



Distancia: 2826



Distancia: 4038



Distancia: 3995

5. Análisis de la Eficiencia.

5.1. Características de la máquina utilizada

Hardware

Procesador: Intel Core i7-3630QM CPU @ 2.40GHz x 8

Memoria: 7,7 GiB

Sistema Operativo: Ubuntu 14.04 LTS 64-bit

5.2. Comparativa entre ambos algoritmos.

	small5	**small7**	**small10**	**small11**	**berlin13**
Backtracking	0,000106	0,000347	0,011381	0,03754	0,345614
Branch & Bound	0,000092	0,000321	0,014288	0,08777	1,893950