Problema del viajante de comercio
Travelling salesman problem

José Antonio Álvarez Ocete - Norberto Fernández de la Higuera
Javier Gálvez Obispo - Yábir García Benchakhtir

2 de mayo de 2018

Índice

1. Descripción del problema

En el problema del viajante de comercio (*Travelling Salesman Problem*) se nos dan las coordenadas de un conjunto finito de ciudades y se nos pide encontrar un recorrido de las misma de forma que la distancia que recorramos sea la menor posible.

2. Soluciones greedy

2.1. Nearest Neighbor

Teniendo como conjunto de ciudades Ω y la matriz $\mathcal{M}$ que contiene la distancia entre cada una de las ciudades, partimos de $\mathcal{C}$ el conjunto de posibles candidatos (su valor inicial es Ω) y el conjunto solución $\mathcal{R}$ (se inicializa vacío).

Mediante ésta estructura el algoritmo se puede clasificar como greedy, teniendo en cuenta que a cada iteración del algoritmo se busca el óptimo local (la ciudad más cercana a la actual) y todos los candidatos usados pasan al conjunto solución $\mathcal{R}$, descartándolos del conjunto $\mathcal{C}$.

El funcionamiento del algoritmo sería el siguiente:

Algorithm 1 Nearest Neighbor

```
R = [random() %| Ω |]
C - R[0]
for pos in [0,n-2] do
    bestPos = nearest(M[R[pos]],C)
    R.push(C[bestPos])
    C.erase(bestPos)
end for
return R
```

Donde la función "nearest"devuelve la posición de la ciudad que, de entre todos los candidatos, está más cerca a la actual.

El algoritmo es O(n^2) ya que tiene la misma estructura que el algoritmo de ordenación por selección, donde se escogía el valor mínimo de las componentes que no se habían insertado todavía. En nuestro caso se escoge la ciudad que es la más cercana a la actual y se incluye en el conjunto solución, a partir del cual se repetirá el mismo procedimiento.

2.2. Cheapest Insertion

Si tomamos Ω el conjunto de ciudades, donde cada ciudad la notamos como ω y $\mathcal{M}$ la matriz simetrica de la distancia entre las ciudades.

Para este tomamos $< i, j >$ tales que $min(M) = M_{ij}$. Partimos de la solución $\Gamma = \{\omega_i, \omega_j\}$ y en cada iteración buscamos la ciudad de índice p tal que $\omega_p \notin \Gamma$ y que minimiza:

$$M_{ip} + M_{jp} - M_{ij}$$

Una implementación en pseudo-codigo sería:

Algorithm 2 Cheap Insert

```
i, j = nodes(min(M))
path = [i, j]
C = Ω - path
while C not empty do
    for city in cities do
        if city not in path then
            for node in path do
                minimizeDistance(node, city, node+1)
            end for
            insert(path, T)
            remove(C, T)
        end if
    end for
end while
return path
```

Este algoritmo es un algoritmo claramente greedy donde:

- Nuestro conjunto de candidatos es el conjunto Ω.
- Mantenemos una lista de candidatos ya usados.
- Como criterio para determinar si Γ es valido buscamos no tener ciclos y que no se repitan los nodos.
- Nuestra función de selección es la definida en algoritmo anterior y que determina en cada situación cual es la menor distancia que podemos agregar.
- La función que nos permite comparar soluciones es la distancia del camino final Γ

2.3. Otro algoritmo posible

Finalmente hemos implementado una tercera estrategia basada en un algoritmo genético. Dadas las dimensiones del problema (NP-completo), es natural buscar una solución lo suficientemente buena aunque no tenga por qué ser la mejor. Este es exactamente el punto de vista que hemos tomado para abordar esta tercera solución.

Los algoritmos genéticos son un tipo de estrategias clasificadas como metaheurísticas. Esto es, utilizadas en ámbitos muy diversos ya que operan de forma abstracta e independiente al problema en si. En particular, los algoritmos genéticos se basan en la teoría evolutiva de Darwin en la que una población, mediante cruces entre sus individuos y mutaciones, es cada vez más fuerte conformen pasan las generaciones. Para más información sobre el tema:

`https://es.wikipedia.org/wiki/Metaheuristica`
`http://sci2s.ugr.es/graduateCourses/Metaheuristicas`
`https://es.wikipedia.org/wiki/Algoritmo_geneticos`

En un algoritmo genético consideramos un conjunto o población de soluciones (también llamados genes o individuos de la población), inicializadas aleatoriamente, y realizamos distintas operaciones sobre ellas para ir mejorándolas y así obtener soluciones cada vez mejores. En nuestro caso, cada individuo será una permutación de números: el orden en el que recorremos las ciudades.

Las operaciones que se realizan sobre los individuos son operadores de cruce y operadores de mutación:

- Los operadores de cruce obtienen uno o varios individuos (hijos) a partir un par de soluciones de la poblacion (padres). En nuestro caso utilizaremos el operador de cruce de orden:

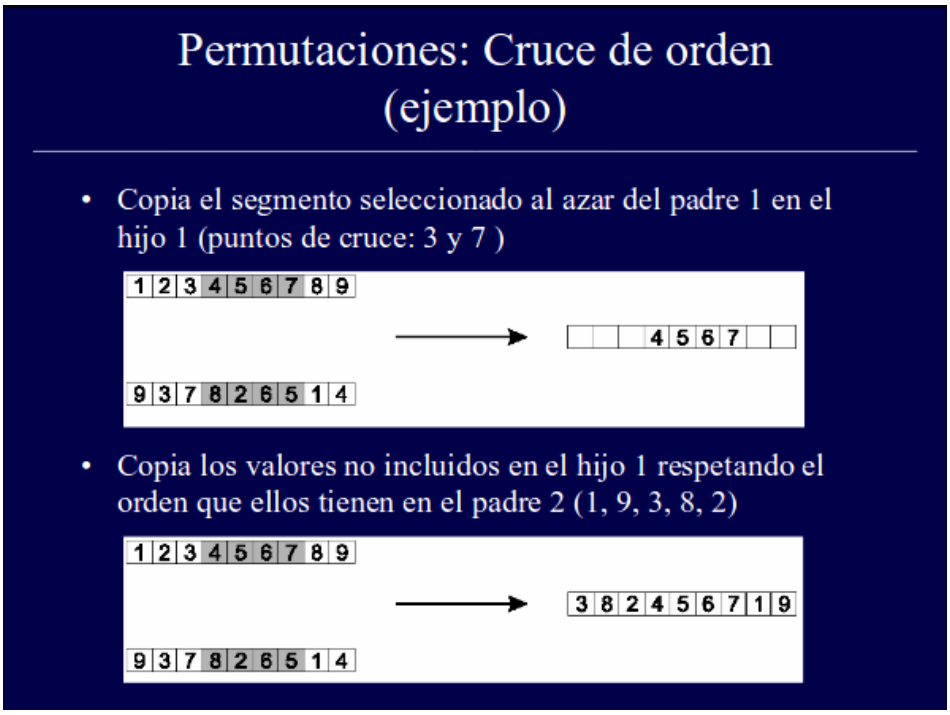

Figura 1: Operador de cruce de orden.

- Los operadores de mutación están limitados por una cierta probabilidad fijada. Es decir, no todos los individuos de una población mutan. Se aplican sobre un único individuo y lo alteran unicamente a él. En nuestro caso utilizaremos el operador de mutación por inserción:

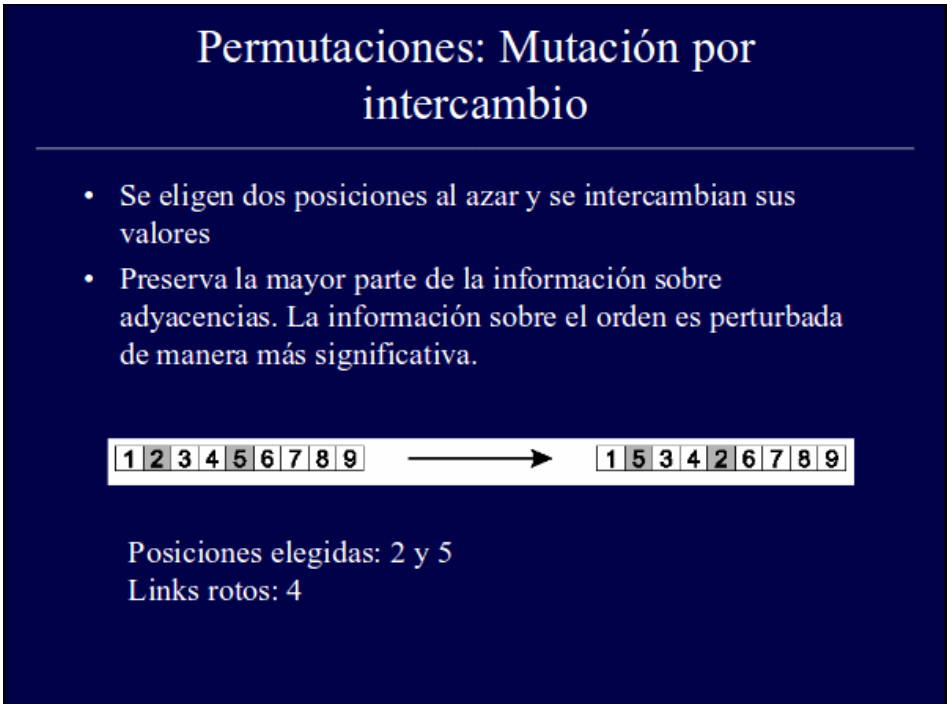

Figura 2: Operador de mutación por intercambio.

Ambos imágenes provienendel siguiente enlace, diapositivas 20 y 17 respectivamente: http://users.exa.unicen.edu.ar/ icom-

pevol/filminasenpdf/terceraclase.pdf

Además, hemos de medir cuan buena es cada una de nuestras soluciones, necesitamos una **función de evaluación**. Esta será la longitud de la ruta determinada por la permutación.

Finalmente y antes de explicar el flujo del algoritmo en si hace falta definir un último operador: el de selección. Este operador define como se realiza la selección de padres de una generación (si cogiesemos unicamente los mejores convergeriamos muy rápido a un óptimo local pues no hay diversidad en nuestra población mientras que si se selecciona de forma puramente aleatoria nos estaríamos teniendo en cuenta cuan buena es cada solución). Aplicaremos **selección por torneo**. En esta estrategia se seleccionan de forma aleatoria un número pequeño de individuos (3 o 5 son valores comunmente utilizados), y de entre estos se escoge el mejor.

Una vez definidos los operadores, el algoritmo en si es sencillo. Realizaremos $n_g eneraciones$ generaciones a partir de una poblacion de N individuos obtenida aleatoriamente. En cada generación se escogen padres (con el operador de selección) y se cruzan (con el operador de cruce) hasta obtener una nueva generación de N elementos. Estos nuevos individuos se mutan (bajo una probabilidad $mutate_p robability$) y finalmente se sustituye la población anterior por la nueva.

```
// Initialize the population and evaluate it
InitializePopulation(population, pob_size, n_cities);
best_gen_ever = *population[0];
EvaluatePopulation(population, best_gen_ever, cities);

for (int i=0; i<n_generations; i++) {
// Obtain the next generation and apply crossover from the elements of population
ApplySelectionAndCrossover(population, new_generation);

// Apply mutation to the new population with a probability prob
ApplyMutation(new_generation, mutate_propability);

// Evaluate the new population and save the best solution found
EvaluatePopulation(new_generation, best_gen_ever, cities);

// Replace the old population with the new one
ReplaceGeneration(population, new_generation);
}
```

Nuestra mejor solución está almacenada en $best_g en_e ver$.