Cómputo Evolutivo
Proyecto 1: Agente Viajero

Andrea Itzel González Vargas
Carlos Gerardo Acosta Hernández

Facultad de Ciencias UNAM
Entrega: 07/09/16

Introducción

Panorama general

Como primer proyecto de la materia implementamos un Algoritmo Genético para resolver instancias del *Problema del agente viajero* (TSP, por sus siglas en inglés), -se consideró el caso simétrico. Para lograrlo, utilizamos el framework de desarrollo escrito en *Java* provisto por el ayudante de laboratorio, Roberto Monroy, en las prácticas de clase.

Dado que el framework provee de una estructura modularizada para el desarrollo, el código pertinente y referente a los principios de un AG -como los procesos de selección, cruzamiento, mutación, así como los operadores que involucran- pueden revisarse independientemente en la carpeta de "sources" (src/) del proyecto.

Como entrada, el programa recibe un archivo con extensión *.tsp*, que representa, con cierta sintáxis especial dentro de un archivo de texto plano, una instancia del problema del agente viajero. En el directorio del proyecto es posible encontrar numerosas instancias[1] del problema, entre ellas las que fueron empleadas para el análisis de resultados -burma14, ulysses16, ulysses22, gr17 y gr21. Todas ellas se pueden revisar dentro del proyecto en el carpeta de datos (tsp/).

De ser necesario, es posible leer instrucciones sobre la ejecución del programa en el *Readme* incluído en la raíz del directorio del proyecto. También, para revisar el contenido sin necesidad de descargarlo, se puede navegar amablemente por las líneas de código desde nuestro repositorio alojado en *GitHub*[2]

[1] Se pueden descargar de la página del curso aquí.
[2] Enlace a repositorio del proyecto: Proyecto-TSP

Especificación

La implementación está basada en la solución al problema propuesta en el libro de *Algoritmos Genéticos* de Kuri-Galaviz.

Codificación del problema

Como se comenta en el libro, hay una marcada preferencia por la utilización de representaciones binarias para la codificación en problemas que intentan resolver los algoritmos genéticos, principalmente porque se ven favorecidas por los resultados del *Teorema del Esquema*. Sin embargo, para este problema en particular tiene un par de desventajas que nos llevaron a elegir otra codificación.

Para empezar, una instancia del problema posee un número fijo de ciudades que tiene que recorrer el agente viajero. Es importante procurar representar cada una y mantener esa representación a lo largo del algoritmo. En caso de elegir una representación binaria, la codificación que se hace de cada ciudad requiere de ensambles binarios tan grandes como sea necesario para representar el total de ciudades.

Por otro lado, ya en la ejecución del algoritmo genético, los operadores de cruza y mutación fácilmente pueden producir ensambles de binarios que en su decodificación no sean parte del dominio del problema, es decir, que si cambiamos uno sólo de los bits quizá deje de existir un mapeo correspondiente con las ciudades de la instancia. No sólo eso, también es posible que dentro de un mismo ensamble de bits de un *genotipo* resulte una ciduad repetida, lo cuál, por definición no está permitido. Se vuelve entonces necesario implementar un corrector que entre en juego luego de la actividad de los operadores y en la generación de una población inicial.

Es por lo anterior que nos decidimos por implementar la **codificación no binaria** descrita en el libro. Para esta codificación cada individuo es un recorrido por las ciudades y cada alelo en su fenotipo es una ciudad, representada por un entero. De esta manera, se asegura que los operadores sólo se manejen sin afectar la validez de un solo alelo como en la representación binaria, entonces sólo tenemos que preocuparnos por la validez del individuo. Además se ajusta bastante bien a las condiciones en que recibimos las instancias de TSP, pues cada ciudad es un nodo con un entero asociado. La generación de una población inicial "aleatoria" se hace cuidando que se generen individuos válidos, asignado sólo una vez cada ciudad a cada nuevo individuo -puede verse de hecho como múltiples permutaciones de la lista de ciudades en la instancia del problema. El resto del proceso del AG se reconsideró bajo las premisas de esta representación no binaria, por lo que los operadores no constituyen un riesgo para la validez de los códigos genéticos ni los individuos que generan y no fue necesaria la implementación de un corrector.

Evaluación de la población y Selección

El problema del agente viajero es un problema de minimización, como no nos es posible utilizar una traducción directa de uno de maximización, pues al multiplicar por -1 la función de evaluación obtendremos costos no permitidos por la definición del problema, hicimos un reajuste a la evaluación de la población. Primero consideramos el fitness como nuestro costo de viaje, es decir, cada individuo representando un viaje tiene un costo asociado que es la suma de las distancias entre sus ciudades, considerando que es un recorrido cíclico, se suma también la distancia entre la última

ciudad y la primera (sin olvidar que la distancia es simétrica en estos problemas). Definida para un genotipo como:

$$fitness(g) = d(g[0], g[n-1]) + \sum_{i=0}^{n} d(g[i], g[i+1]) \quad (1)$$

Por conveniencia de implementación bajo el framework -pues una implementación de la interfaz de *FitnessFunction* emplea su evaluación sobre un sólo fenotipo- se decidió que ésta fuera el fitness, aunque se asocie más fácilmente con el concepto de *función objetivo*.

Por otro lado, la función objetivo implementada y que para la lógica de nuestro AG lo tratamos como función de adaptación -en el framework se aplica para una población y por tanto resultaba más útil puesto que el grado de adaptación de un individuo depende del resto de la población-, la definimos para un genotipo, dados los costos máximo y mínimo de la población, como:

$$evalObj(g) = (fitness_{max} + fitness_{min}) - fitness(g) \quad (2)$$

Señalamos que el FrameworkAG considera a los mejores y peores individuos para un problema de maximización, por lo que aunque se señale en la interfaz gráfica del programa una acotación verde para los mejores individuos en las estadísticas, en realidad el individuo que nos interesa a nosotros es el de acotación color rojo, pues es el de menor costo de viaje en la población.

Decidimos utilizar este remapeo de la función de adaptación con el fin de utilizar la **selección proporcional** de "ruleta". La elegimos por nuestra experiencia con la práctica anterior.

Cruza

Para el operador de cruza, implementamos el **cruzamiento uniforme ordenado** descrito en el libro del profesor. Con este tipo de cruzamiento nos aseguramos de que se generen individuos válidos al aplicar el operador sobre dos individuos de la población que hayan sido seleccionados para este proceso. Es decir, el nuevo individuo no tendrá repeticiones de ciudades.

Para lograr esto se emplea una "máscara" de bits del tamaño del genotipo. Como en el cruzamiento uniforme, se realiza un experimento de *Bernoulli* por cada ciudad contenida en el individuo que se reflejará en la máscara. Para aquellos índices donde la máscara tenga valor de 1 se agregarán las ciudades correspondientes del primer padre al nuevo individuo (en consecución) y aquellos índices donde la máscara haya resultado con valor 0, se generará una lista de ciudades correspondientes a esos ceros del primer padre y se agregarán al nuevo individuo sobre los genes por asignar en el orden en que aparecen esas ciudades dado el genoma del segundo padre.

Lo anterior es un mero esbozo de explicación que puede ser encontrado con más detalle en el libro[3] y puede revisarse con detalle nuestra implementación en el archivo de clase *AVCrossover.java*.

Mutación

De la misma manera que con el cruzamiento, el operador mutación está planteado para producir un individuo "mutado" válido en la instancia del problema.

[3]Kuri-Galaviz, Algoritmos Genéticos, págs. 91,92

El algoritmo utilizado para esta mutación también hace uso de una "máscara", una cadena de bits del mismo tamaño del individuo a mutar. Ésta se genera dada la probabilidad de mutación p en la ejecución del programa. Se establecerá entonces un bit 1 en la máscara con probabilidad p y un 0 con probabilidad $1 - p$.

Posteriormente, para aquellos índices en que la cadena de bits tenga asignado un 1, se generará una lista con las ciudades correspondientes del individuo y se eliminará ese gen del individuo, después se realizará una permutación de los elementos de esta lista de manera que ninguno termine en la posición en que se encontraba originalmente al crearla.

Finalmente, en los genes vacíos del individuo se acomodarán las ciudades en el orden que aparecen en la lista temporal que se permutó. De esta forma, no es posible repetir ciudades, pues la mutación es básicamente una permutación de las ciudades en el individuo original a mutar.

Para revisar el pseudo-código del algoritmo, igualmente puede ser encontrado en el capítulo 3 del libro y nuestra implementación en el archivo de clase *AVMutation.java*, dentro del directorio de código fuente del proyecto.