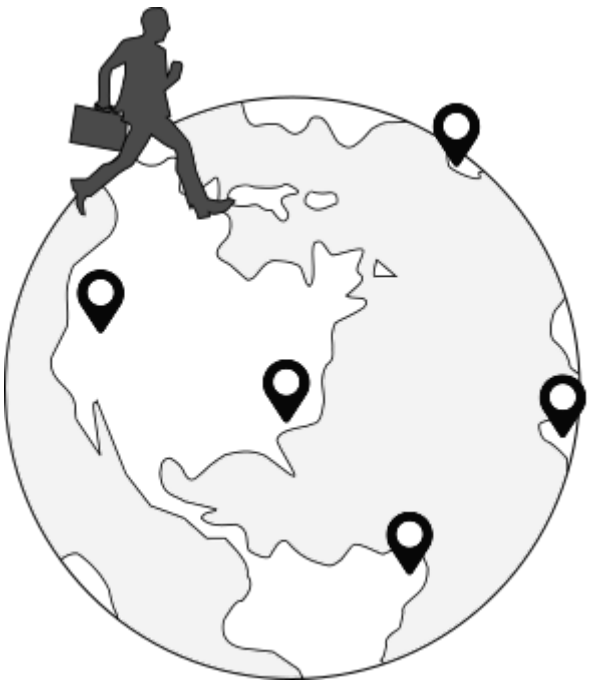
Problema del viajante

Algoritmo Genético

Por: Ander Monreal Ayanz



Índice

¿Qué es un Algoritmo Genético?.....3

¿Cuál es el problema del viajante?.....4

Puntos importante a tratar.....5

Implementación..... 6

1. Método de obtención de la población inicial.....7
2. Método de asignación de fitness.....8
3. Métodos de selección de progenitores.....8
    3.1 Método del torneo..... 8
    3.2 Método de la ruleta..... 9
4. Métodos de cruzamiento..... 9
    4.1 Cruzamiento por ciclos.....10
    4.2 Cruzamiento parcialmente mapeado..... 10
5. Método de obtención de mutaciones.....11
    5.1 Mutación por intercambio..... 11
    5.2 Mutación por inserción..... 11
    5.3 Mutación por sacudida..... 12
6. Método de selección de supervivientes..... 12

Conclusión..... 14

¿Qué es un Algoritmo Genético?

Un algoritmo genético (AG) es una técnica de optimización y búsqueda inspirada en la evolución biológica y los procesos genéticos naturales. Utilizan una población de soluciones potenciales (llamadas individuos o cromosomas) que evolucionan a lo largo de generaciones mediante operaciones inspiradas en la genética y la selección natural, como la selección, el cruce (crossover) y la mutación. El proceso de evolución se realiza en varias etapas iterativas (generaciones), y las soluciones mejores adaptadas tienen más probabilidades de sobrevivir y reproducirse.

Los Algoritmos Genéticos (AG) son una parte de la COMPUTACIÓN EVOLUTIVA. Son algoritmos de búsqueda adaptativa y heurística basada en las ideas de la selección natural y la genética. HEURÍSTICA: Conjunto de reglas no necesariamente formalizadas o rigurosas que conducen a la resolución de un problema. Los AG hacen un uso inteligente de las búsquedas llevadas a cabo en problemas de optimización. Los AG, aunque son aleatorios, hacen uso de la información acumulada durante la búsqueda pasada para orientar la búsqueda en el futuro.

Los algoritmos genéticos se utilizan en una amplia variedad de aplicaciones, incluyendo optimización de funciones matemáticas, diseño de ingeniería, programación automática, aprendizaje automático, y muchos otros campos donde se requiere encontrar soluciones óptimas o cercanas a óptimas en problemas complejos y multidimensionales. Su capacidad para explorar el espacio de búsqueda y encontrar soluciones eficientes los hace muy versátiles y efectivos en muchas áreas de la ciencia y la ingeniería.

¿Cuál es el problema del viajante?

El Problema del Viajante (Traveling Salesman Problem, TSP) es un problema clásico de optimización combinatoria que se adapta bien a los algoritmos genéticos debido a su complejidad y naturaleza de búsqueda en un espacio de soluciones vasto.

En el Problema del Viajante, se busca determinar la ruta más corta posible que visite todas las ciudades (nodos) exactamente una vez y regrese al punto de partida (ciclo hamiltoniano). Cada ciudad está conectada por carreteras (aristas) con ciertas distancias o costos asociados. El objetivo es encontrar la permutación de ciudades que minimice la distancia total recorrida por el viajante.

Los algoritmos genéticos ofrecen una forma poderosa de abordar el Problema del Viajante debido a su capacidad para manejar problemas de optimización combinatoria y su habilidad para explorar y explotar soluciones en un espacio de búsqueda complejo.

En este caso, el objetivo es recorrer algunas de las ciudades de España, con el fin comentado anteriormente.

Puntos importante a tratar

Es importante recalcar el funcionamiento del algoritmo, ya que el algoritmo cumple con las especificaciones. Se puede cambiar perfectamente la ciudad de inicio y fin, y también se puede ajustar el número de ciudades con las que trabajar.

Este algoritmo, sí que alcanza una solución, tal vez no sea la más óptima, esto dependerá primordialmente del número de iteraciones que le queramos dar a nuestro algoritmo. Por cada iteración nuestra población irá mejorando (el "irá mejorando" se explicará después) y tendrá mejores individuos que se acercarán más a la solución más óptima. Aparte del número de iteraciones, también tiene más factores que hacen el algoritmo más eficiente, se comentarán después.

En cuanto al coste temporal, la verdad que estoy muy contento con el resultado ya que pese a no haber usado ningúna inteligencia artificial en ningún momento y, haberle añadido un pequeño tiempo de espera al principio para hacerlo más agradable a la vista, el algoritmo a resultado tener un coste temporal muy pequeño e incluso para un número de iteraciones elevadas, teniendo en cuenta que prueba diferentes tipos de métodos que se comentarán posteriormente.

Como ya he comentado previamente, he repasado los puntos exigidos en la entrega de esta práctica y cumple perfectamente con todo.

Implementación

La implementación del programa ha sido realizada en Python, exactamente en el programa "main.py". El código está bastante comentado a la vez que verboso para que sea más comprensible a la hora de examinar, se han marcado los métodos en el programa.

Se ha utilizado la librería de "pwn" para mostrar tanto el progreso como el resultado final de forma más bonita, si no funciona esta librería se puede instalar con "pip install pwntools" o "python3 -m pip install pwntools". **Si hay algún problema con esto, mandame un correo y te quito toda esta parte**, que solo es para que se imprima más bonito el código.

El algoritmo trabaja con una población de tamaño 10 (Este parámetro se puede cambiar), una ciudad inicial la cual puede ser variable a elección del usuario, además puede trabajar con diferentes tamaños de ciudades. Para todo esto, el algoritmo abarca varios métodos necesarios para hallar una solución ciertamente óptima a la vez que eficiente. Estos métodos empleados son los siguientes:

1. Método de obtención de la población inicial

Este método, es el encargado de devolver una población de individuos inicial, por defecto está seteado a 10 individuos, pero se puede cambiar perfectamente sin romper el funcionamiento. Se haría simplemente cambiando la variable global "TAM_POBLACION".

Un individuo es una posible solución la cual se genera aleatoriamente evitando repeticiones de ciudades, **estos individuos serán los cromosomas de mi Algoritmo Genético,** todos ellos tendrán el mismo formato siendo en la casi plenitud diferentes entre ellos y representados en una lista.

Esta función tiene un parámetro de entrada "ciudad_inicial" que corresponde a la ciudad tanto de inicio como de final, de esta forma la ciudad inicial y final puede ser variable a elección por el usuario.

Por último, se devuelve una lista con todas las ciudades que aparecen en la variable global "CIUDADES" aleatoriamente, siendo el inicio y el fin de la lista la "ciudad_inicial".

Ejemplo de cromosoma (individuo):

['Pamplona', 'Donostia', 'Bilbao', 'Oviedo', 'Coruna', 'Leon', 'Zaragoza', 'Girona', 'Barcelona', 'Tarragona', 'Valencia', 'Alicante', 'Murcia', 'Cordoba', 'Cadiz', 'Sevilla', 'Huelva', 'Caceres', 'Toledo', 'Madrid', 'Pamplona']

2. Método de asignación de fitness

A continuación, pasamos a una de las funciones más importantes del código, "calcular_costo_ruta()". Esta función **es la encargada de calcular lo que va a ser mi fitness para todo el Algoritmo Genético.**

Como argumento de entrada, recibe la población obtenida en el método anterior y la ciudad inicial seteada por el usuario. Se respalda de las variables globales "CIUDADES" y "M". Con todo lo anterior, **calcula el costo de cada individuo haciendo el sumatorio de costes de cada ciudad de la lista del cromosoma con la siguiente ciudad.** De esta forma obtengo la distancia total para un individuo. Por último, divido uno entre cada coste de cada individuo para trabajar con decimales y todos los resultados los meto a una lista que posteriormente devuelvo y que **contiene el fitness de cada individuo de la población.**

3. Métodos de selección de progenitores

Una vez tenemos los fitness de los cromosomas de la población, es hora de elegir los padres, para ello he decidido usar el método del torneo y el método de la ruleta, posteriormente a explicar ámbos, haré una pequeña comparativa. El algoritmo, en todas las iteraciones probará a utilizar ámbos métodos

3.1 Método del torneo

Para este método, se le pide al usuario que introduzca un valor entero que lo llamaré 'k'. Este valor, el cuál no será ni menor que 0 ni mayor que el tamaño de la población, determinará **'k' cromosomas aleatorios escogidos**

de la población los cuales se enfrentarán entre ellos para determinar un ganador, este ganador será el que tenga **el fitness más alto** calculado con el método anterior. Contra mayor sea este número entero, más posibilidades habrá de escoger el mejor fitness.

3.2 Método de la ruleta

Para este segundo método, se realizará **un sorteo aleatorio entre todos los individuos de la población**, como el fitness de los individuos se asemeja considerablemente entre sí, he decidido asignarles a **todos los cromosomas la misma probabilidad** de victoria.

Como conclusión entre estos dos métodos, se puede observar después de realizar varias pruebas a la vez que lógicamente, que **el método del torneo es mucho más probable que obtenga un padre con un fitness mejor que el padre resultante del método del torneo**, esto se debe a que el método del torneo intenta obtener de los seleccionados el individuo con mejor fitness.

4. Métodos de cruzamiento

En este punto, he creado dos métodos de cruzamiento, el cruzamiento por ciclos y el cruzamiento parcialmente mapeado. Ambos métodos, dependen de dos padres los cuales serán obtenidos en el punto anterior con el método del torneo y el método de la ruleta.

Para cada cruzamiento se probará con los diferentes padres es decir, se obtendrán 4 hijos en total, 2 de ellos resultan por el cruzamiento por ciclos donde el primer hijo será producido con dos padres ganadores del método del torneo y el segundo hijo será con dos padres ganadores del método de la ruleta, para los otros dos hijos se usarán los mismos pares de padres comentados anteriormente

pero esta vez utilizando el método de cruzamiento parcialmente mapeado.

4.1 Cruzamiento por ciclos

Este cruzamiento es bastante complejo de explicar mediante texto. Básicamente, la idea es dividir el cromosoma en ciclos, es decir, subconjuntos de elementos tales que cada uno de sus elementos siempre aparece emparejado con otro elemento del mismo ciclo cuando los padres se alinean. Este método pretende preservar información sobre la posición absoluta de los elementos dentro de cada cromosoma progenitor.

De todas formas, he intentado hacer el código de Python bastante verboso con nombres de variables y métodos entendibles. Este método devuelve un hijo resultante del cruzamiento por ciclos.

4.2 Cruzamiento parcialmente mapeado

Al igual que el anterior este cruzamiento es complejo de explicar mediante texto. El código de Python queda bastante verboso también. Es importante recalcar que en este caso no toda la información presente en los progenitores se preserva.

La diferencia principal entre el cruce por ciclos y el cruce parcialmente mapeado radica en cómo combinan la información de los padres para generar descendientes y cómo manejan la preservación de estructuras en las soluciones. Donde **el cruce por ciclos sí que preserva información sobre la posición de los elementos.**

5. Método de obtención de mutaciones

Para realizar las mutaciones, me respaldo de la librería random, la cual voy a tener que usar para escoger aleatoriamente 2 números y operar con ellos.

Para hacer el algoritmo más completo he decidido implementar 3 métodos de mutación diferentes para obtener más posibilidades. Cada tipo de mutación será probada con cada hijo resultante de los cruzamientos anteriores, es decir en total obtendremos 24 mutaciones diferentes, 8 serán por el método de intercambio, otras 8 por el método de inserción y las últimas 8 por el método de sacudida.

Gracias a este proceso puedo obtener diferentes variedades de cromosomas los se acercarán a una solución bastante óptima.

5.1 Mutación por intercambio

Consiste simplemente en escoger aleatoriamente dos elementos del hijo e intercambiarlos de posición. Puede ser que mute un elemento que esté en una posición bastante favorable o, que mute un elemento que esté en posición desfavorable.

5.2 Mutación por inserción

Consiste en escoger aleatoriamente dos elementos del hijo y adelantar el segundo elemento justo detrás del primer elemento corriendo así un elemento para atrás. Al igual que el anterior puede ser que la mutación no sea beneficiosa.

5.3 Mutación por sacudida

Consiste en escoger aleatoriamente dos elementos y, aleatoriamente reordenar todos los elementos existentes entre los dos elementos escogidos aleatoriamente. Al igual que los dos anteriores se puede dar el caso fácilmente que la mutación tenga un fitness peor que el hijo mutado.

Todas estas mutaciones nos producen resultados totalmente válidos, el único problema es que al ser aleatorio, no puedo dictaminar una mutación mejor que otra ya que depende bastante tanto del azar a la hora de escoger las posiciones que se van a editar tanto como del hijo obtenido en el paso anterior del cruzamiento.

6. Método de selección de supervivientes

Para este método, he optado por realizar una selección reemplazando basándose en el fitness de las mutaciones anteriores. Este proceso se encarga de obtener las 24 mutaciones anteriores realizadas y, hacer una selección de las dos mejores mutaciones con fitness más alto.

Una vez tenemos las dos mutaciones con el fitness más alto, crea una nueva población con los mejores 10 individuos de las dos mutaciones y los 10 individuos de la población anterior.

Esta función es también muy importante ya que obtiene por cada iteración poblaciones con mejores fitness, lo que provoca que a la larga es decir, a mayor iteraciones haga nuestro Algoritmo Genético, mejores individuos obtendremos.

7. Condición de salida del Algoritmo

Cuando el algoritmo cumpla con el número de iteraciones determinadas por el usuario, dejará de hacer todo este proceso comentado previamente, y pasará a elegir al individuo con el mejor fitness de la última población de supervivientes resultante.

Como ya he comentado antes, igual no es la solución más óptima pero sí que se acerca bastante para un número de iteraciones mayor que 20.

Este es el resultado final del programa:

Conclusión

Como conclusión final, estoy bastante orgulloso con mi trabajo porque como ya he dicho, no he empleado ninguna inteligencia artificial para realizar el Algoritmo y me ha ayudado muchísimo a entender este tipo de algoritmos, gracias también a los apuntes cogidos en clase he podido realizar esta práctica.

En cuanto a la eficiencia del algoritmo, creo que este algoritmo es bastante eficiente gracias a 3 factores principales:

El primero de ellos es el número de iteraciones, ya que contra más grande sea este número, gracias al método escogido a la hora de seleccionar los supervivientes, me facilita notablemente una mejora en la población.

En segundo lugar, igual no a tanto nivel como el punto anterior, creo que es muy importante el uso del método del torneo ya que usando un valor 'k' ciertamente alto, podemos seleccionar un individuo un fitness beneficioso.

Por último, también cobrando una importancia notoria, creo que el correcto uso de una función que calcule el coste de ruta para usarlo como fitness, ha sido un total acierto, ya que esto permite descartar con mucha mayor facilidad todas aquellas rutas que se alejan del objetivo.