
Universidad de
SanAndrés

PARADIGMAS DE PROGRAMACIÓN

TRABAJO FINAL

**Algoritmos Genéticos - Problema del viajante**

**Alumnas:** Josefina Dehan, Lucia Fernandez Vincent, Nataly Sol Hofkamp, Martina Tahta Dadourian

**Profesores:** Patricio Moreno, Emilio Oca

Segundo cuatrimestre 2022

## Descripción e interpretación propia del problema y la solución propuesta

**Análisis del problema, requerimientos y diseño inicial:**

El problema presentado fue el problema del viajante. Este es un problema reconocido por gran parte de la comunidad de programadores. Dado ciertas ciudades y las distancias entre ellas, se busca encontrar un único camino que sea lo más corto posible y pase por todas las ciudades. Además, se debe pasar por cada ciudad una sola vez y debe regresar a la ciudad donde se comenzó el paseo.

Para esto, desarrollamos un algoritmo genético, un algoritmo que se utiliza para seguir una serie de pasos y encontrar la solución a un problema específico. Los algoritmos genéticos fueron desarrollados por un par de científicos con el fin de optimizar una búsqueda global.
Los algoritmos genéticos simulan el proceso de selección natural desarrollado por el científico Charles Darwin. Esta teoría muestra cómo ciertas especies cambian con el pasar del tiempo, y se pueden llegar a desarrollar tanto que vuelven a las especies irreconocibles.

El proceso de selección empieza con una generación de la cual solo sobreviven los más aptos. De aquí se desarrolla la siguiente generación, con los mejores individuos de la misma. Para clasificar a los mejores, se utiliza un criterio, el cual dependerá del problema específico, este criterio se lo suele denominar **fitness** de un individuo. Teniendo en cuenta este criterio, se implementan diferentes métodos de selección y mutación para que así los individuos varían.

Volviendo a nuestro problema, contamos con los individuos que serían las rutas, que tienen genes (ciudades) y las generaciones que serán un conjunto de rutas (población). A la vez, consideramos a la distancia total de una ruta como su fitness, es decir, cuanto menor sea la distancia, mejor es la ruta, la solución.

Para esto realizamos un lector de archivos, el cual leerá las ciudades de un archivo de texto y las utilizará para el algoritmo. Se buscará la mejor ruta entre dichas ciudades la cual cumpla las condiciones establecidas.

Para el algoritmo en si, implementamos diferentes clases para poder resolver este problema. Principalmente tenemos la clase City que cuenta con diferentes métodos que nos sirven para

poder manejar a las ciudades. Estos métodos nos permiten igualarlas, imprimirlas, calcular la distancia entre dos ciudades y obtener su coordenada x e y.
Además, implementamos la clase Route que cuenta con dos atributos los cuales son un vector de ciudades y su tamaño que dependerá de la cantidad de ciudades. Esta clase cuenta con diferentes métodos útiles como el AddCity que agregara ciudades a la ruta. Además hay un método SwapCity el cual cambiará una ciudad por otra, dependiendo las posiciones que se les pase. Esto nos ayuda a mutar la ruta y a la vez iremos calculando la distancia de la ruta para después saber que tan mejor es a las otras.

También contamos con la clase Population. En esta clase contamos con distintos métodos que imitarán el proceso de selección natural. Primero empezamos con la creación aleatoria de una generación. Luego buscamos hacer una selección para encontrar a las mejores rutas para después usar esas rutas para la reproducción. Antes queremos asegurarnos que las mejores rutas de las ya seleccionadas tengan más chances de ser elegidas al momento de reproducirse. Después ahí seguimos con el método de reproducción el cual reproducirá entre dos rutas padres a una nueva ruta, y se agregará a la nueva generación. Este método de reproducción agarra dos posibles padres y se fija cual tiene mejor fitness. Del que tiene mejor agarra el primer elemento, es decir la primera ciudad y los inserta en el hijo.

De los dos padres se fija cuál tiene un anterior o siguiente elemento a menor distancia al ya elegido para el hijo (es decir los extremos de las puntas sin contar al ya elegido). Por ejemplo, si ya se insertaron los primeros dos elementos del hijo, para el siguiente se buscará en los primeros y en los terceros elementos de los padres ya que son el anterior y el siguiente al segundo. Así va buscando las mejores opciones para las siguientes ciudades hasta que el hijo ya contenga a todas las ciudades de la ruta. Además, se va fijando que la ciudad no esté ya en el hijo para evitar que se repita por algún error.

Después de esto queremos que algunas de estas rutas muten dependiendo del porcentaje de mutación que será generado de manera aleatoria, ya que es este proceso el que reduce las chances de que el algoritmo se estanque en una solución por falta de variación. Esta mutación se basa en que dos ciudades aleatorias cambien de lugar. También implementamos elitismo en dichas generaciones para que se mantengan las mejores rutas entonces la primera ruta de cada nueva generación será la mejor ruta de la generación anterior.

La cantidad de generaciones dependerá de lo que ingrese el usuario que sea el máximo que se generen o el umbral de la diferencia entre los fitness de dos generaciones.

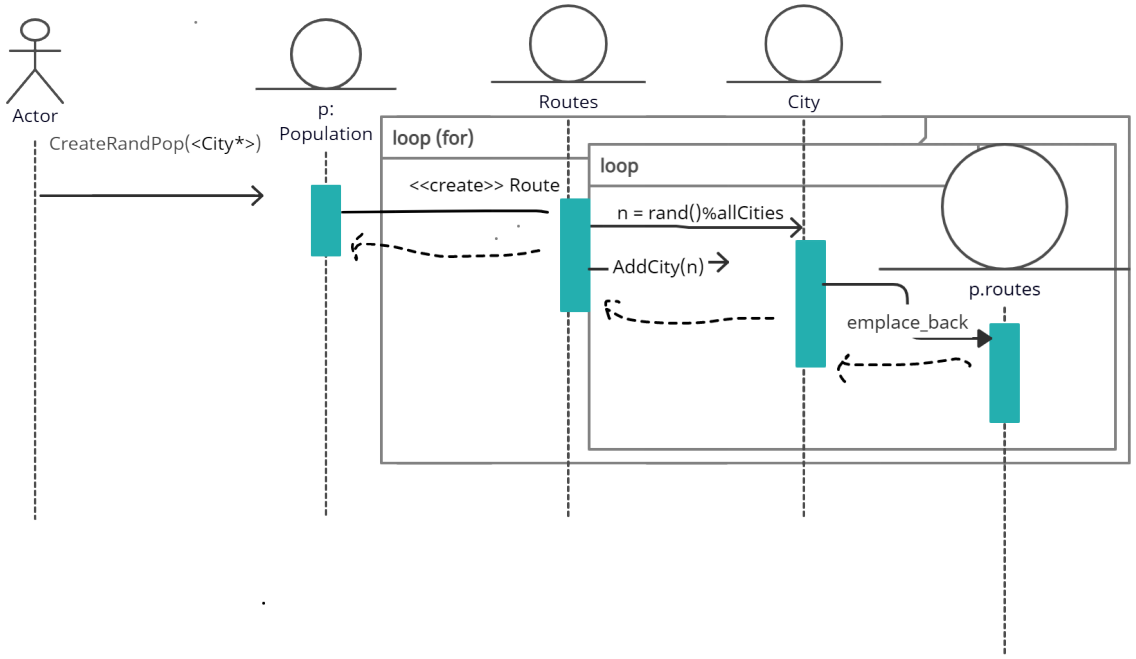
Además, contamos con la clase Graph la cual se usará para generar los gráficos a través del lenguaje python. Contaremos con distintos tipos de gráficos por lo que tendremos distintas clases, una para cada gráfico, así utilizaremos el concepto de polimorfismo. Utilizando métodos de population, guardaremos en archivos la información para los diferentes gráficos:

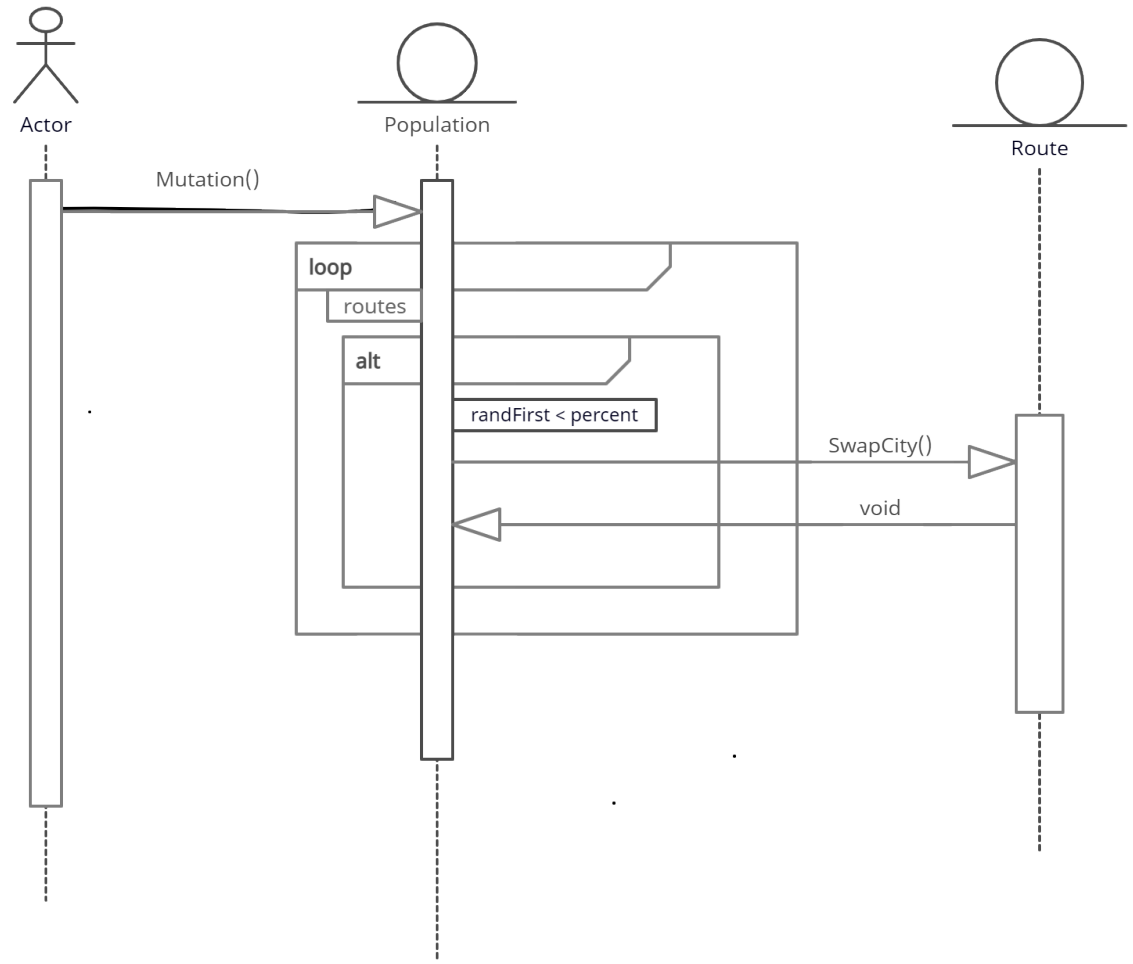
- ruta con mayor fitness
- ruta con menor fitness
- promedio
- mejor ruta de todas

También, contamos con los archivos Reader.cpp y Parser.cpp. El primero lo utilizamos para la lectura de ciudades que se le pasan. El segundo lo utilizaremos para pasar por terminal los parámetros para correr el código.

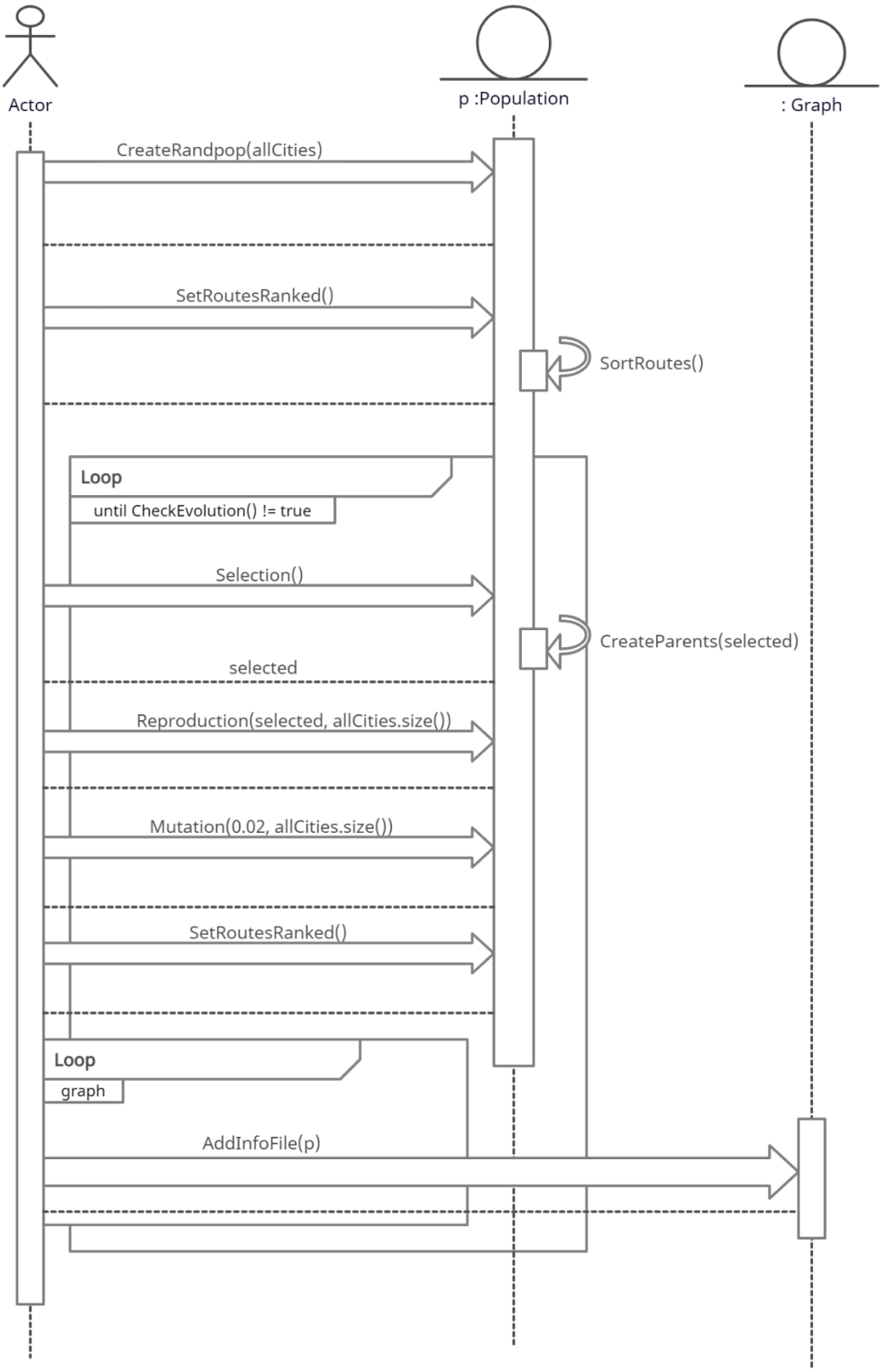
Por último, incluimos el archivo DataGraphs.py para poder ver la información de los archivos en gráficos. Se deben dejar en el main las funciones que crean los gráficos que se hayan pedido que cree el programa.

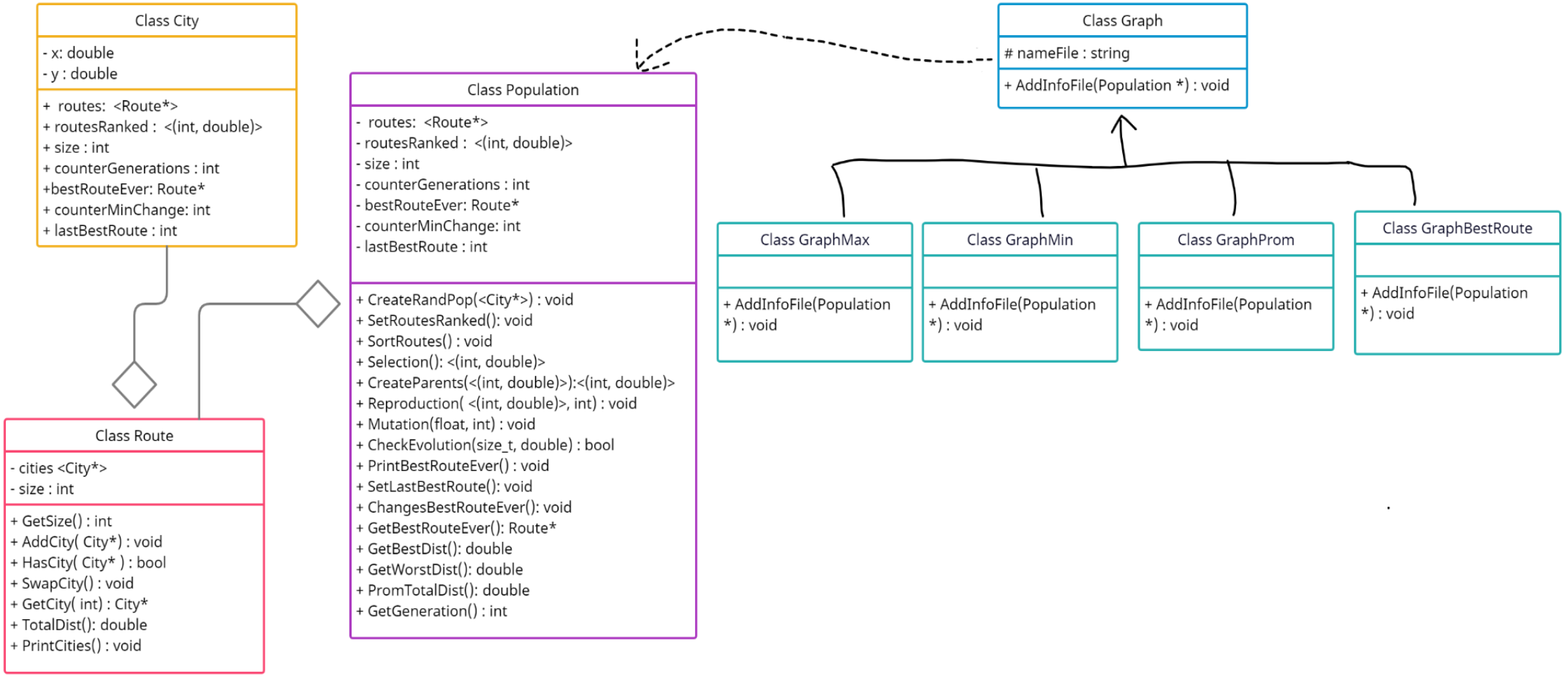
**Diagramas de clases y secuencias:**



Evolución de generaciones:




Class City
- x: double
- y : double
+ routes: <Route*>
+ routesRanked : <(int, double)>
+ size : int
+ counterGenerations : int
+bestRouteEver: Route*
+ counterMinChange: int
+ lastBestRoute : int
Class Route
- cities <City*>
- size : int
+ GetSize() : int
+ AddCity( City*) : void
+ HasCity( City* ) : bool
+ SwapCity() : void
+ GetCity( int) : City*
+ TotalDist(): double
+ PrintCities() : void
Class Population
- routes: <Route*>
- routesRanked : <(int, double)>
- size : int
- counterGenerations : int
- bestRouteEver: Route*
- counterMinChange: int
- lastBestRoute : int
+ CreateRandPop(<City*>) : void
+ SetRoutesRanked(): void
+ SortRoutes() : void
+ Selection(): <(int, double)>
+ CreateParents(<(int, double)>):<(int, double)>
+ Reproduction( <(int, double)>, int) : void
+ Mutation(float, int) : void
+ CheckEvolution(size_t, double) : bool
+ PrintBestRouteEver() : void
+ SetLastBestRoute(): void
+ ChangesBestRouteEver(): void
+ GetBestRouteEver(): Route*
+ GetBestDist(): double
+ GetWorstDist(): double
+ PromTotalDist(): double
+ GetGeneration() : int
Class Graph
# nameFile : string
+ AddInfoFile(Population *) : void
Class GraphMax
+ AddInfoFile(Population *) : void
Class GraphMin
+ AddInfoFile(Population *) : void
Class GraphProm
+ AddInfoFile(Population *) : void
Class GraphBestRoute
+ AddInfoFile(Population *) : void

## Indicaciones para ejecutar correctamente el programa y las pruebas:

Se deben correr todos los archivos cpp involucrados:

*g++ Population.cpp Reader.cpp Route.cpp City.cpp Graph.cpp Start.cpp Parser.cpp main.cpp -std=c++20*

luego, para correr la compilación, se deben ingresar los siguientes argumentos:

*-f o --file <nombre archivo con ciudades>*

*-g o --gen x*

*-u o --umbral y*

y se pueden agregar los siguientes para definir la salida:

*-o o --out z*

*-b o --best*

el **x** será el número de generaciones

el **y** sera el numero del umbral diferencial entre generaciones

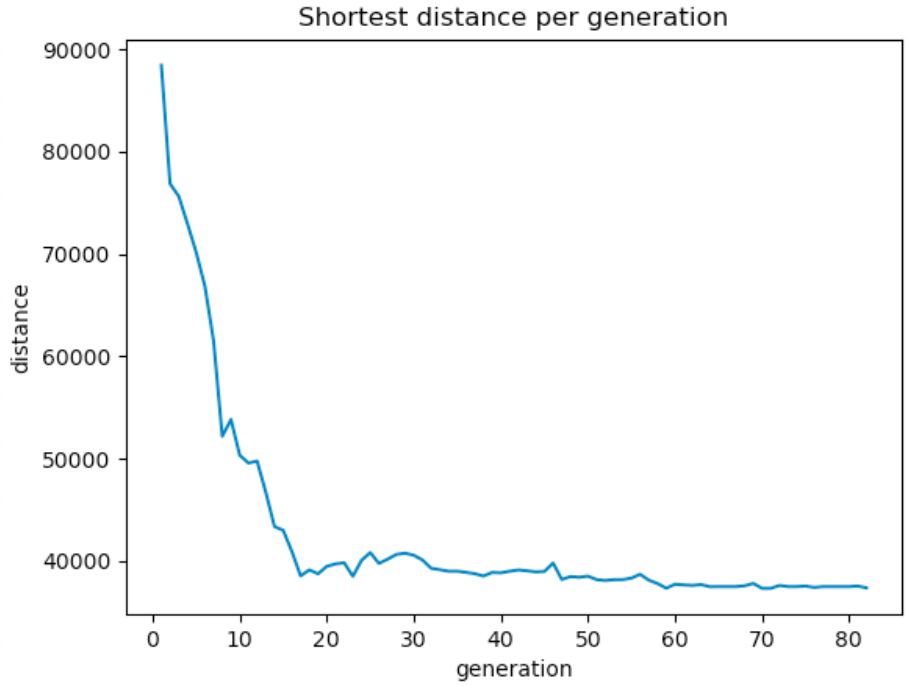
la **z** sera el numero que indica la salida deseada, 1 indica que se quiere el archivo con mayor fitness de cada generación, 2 que se quiere el archivo con la menor fitness de cada generación, 3 que se quiere el del promedio del fitness de cada generación y 4 que se quiere el archivo con las coordenadas en orden la mejor solución encontrada en todo el programa (si se ponen varios juntos, como 1234, se crean los archivos de todos los pedidos, en este caso serían los cuatro)

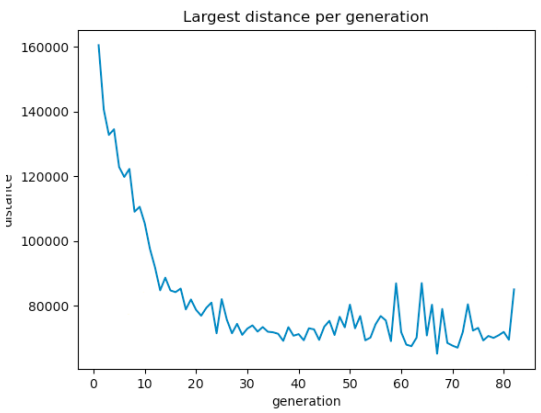
incluir -b implica que se quiere que se imprima la mejor solución encontrada en la terminal

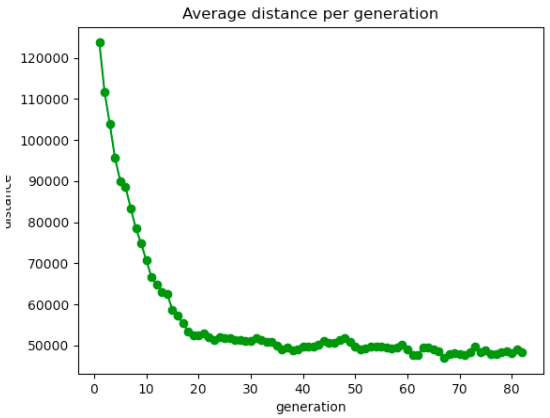
**Resultados de ejecuciones:**

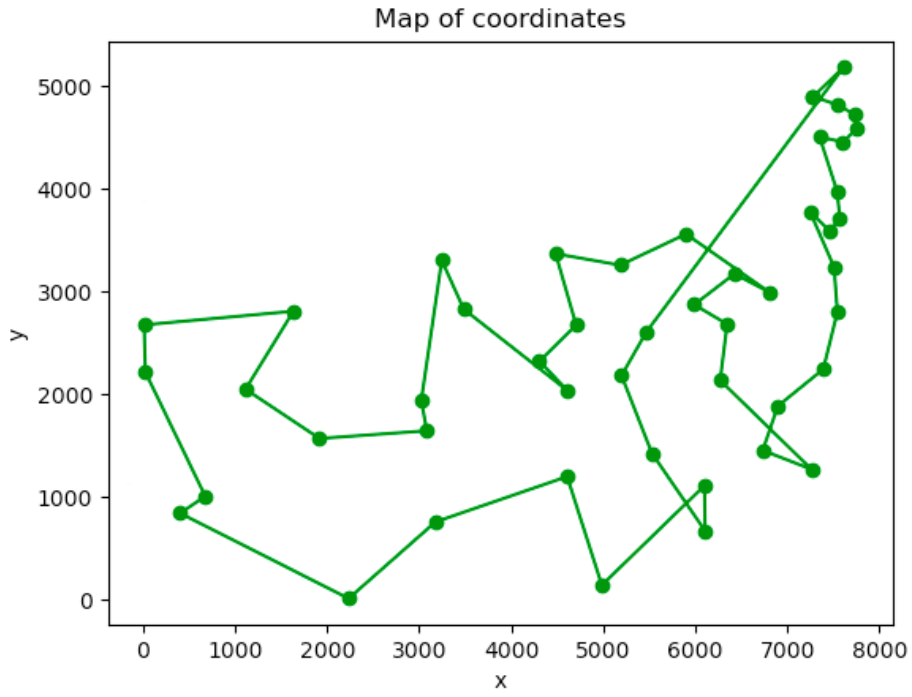
```
(base) Cristians-MacBook-Air:src martina$ g++ Population.cpp Route.cpp City.cpp Reader.cpp Graph.cpp Parser.cpp Start.cpp main.cpp -o p -std=c++20
(base) Cristians-MacBook-Air:src martina$ ./p -f cities.txt -g 500 -u 200 -o 1234
(base) Cristians-MacBook-Air:src martina$ python -u "/Users/martina/paradigmas_programacion/TP-2-3/src/DataGraphs.py"
```

En esta ejecución se frena por alcanzar el umbral por 10 generaciones seguidas. Gráfico de los archivos pedidos:



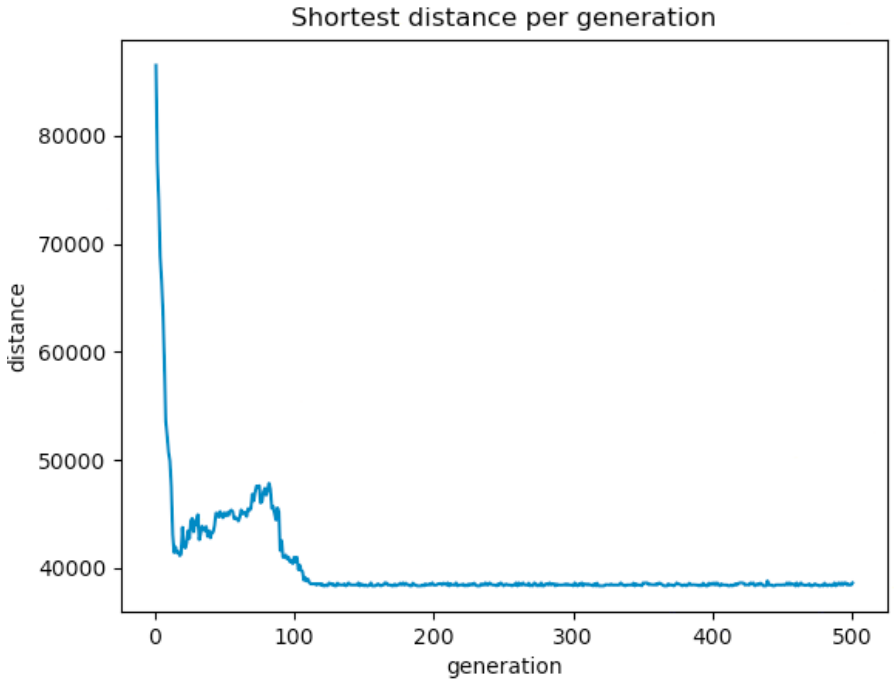



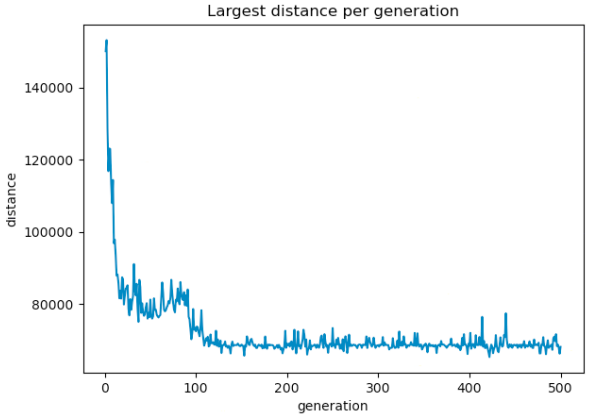



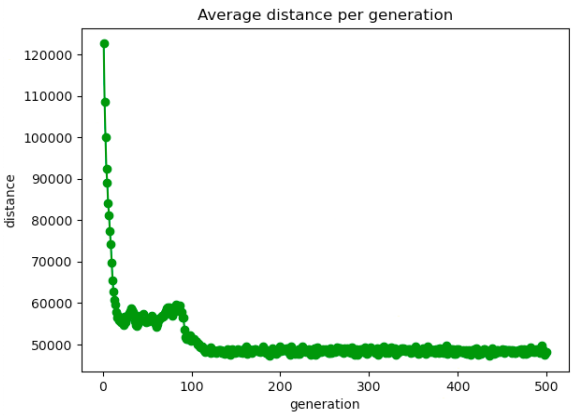


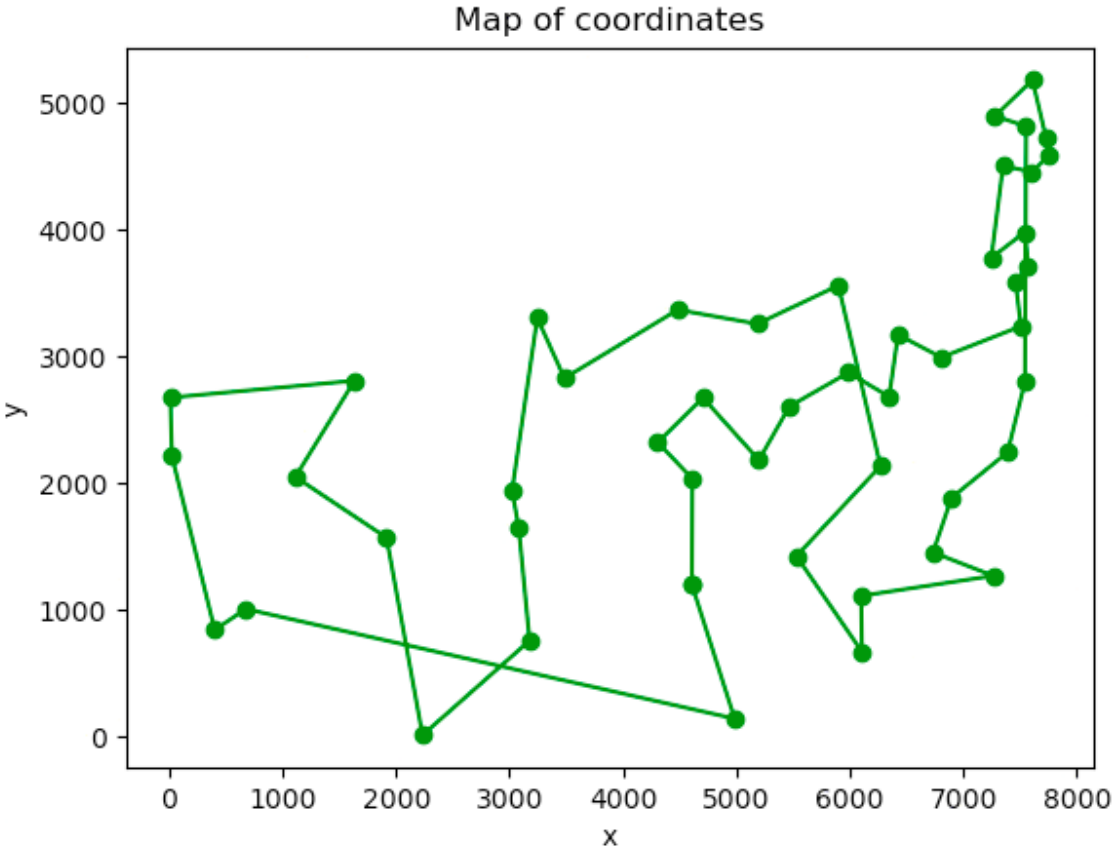

En esta ejecución se frena por alcanzar las 500 generaciones. Gráfico de los archivos pedidos:

```
(base) Cristians-MacBook-Air:src martina$ g++ Population.cpp Route.cpp City.cpp Reader.cpp Graph.cpp Parser.cpp Start.cpp main.cpp -o p -std=c++20
(base) Cristians-MacBook-Air:src martina$ ./p -f cities.txt -g 500 -u 50 -o 1234
(base) Cristians-MacBook-Air:src martina$ python -u "/Users/martina/paradigmas_programacion/TP-2-3/src/DataGraphs.py"
```



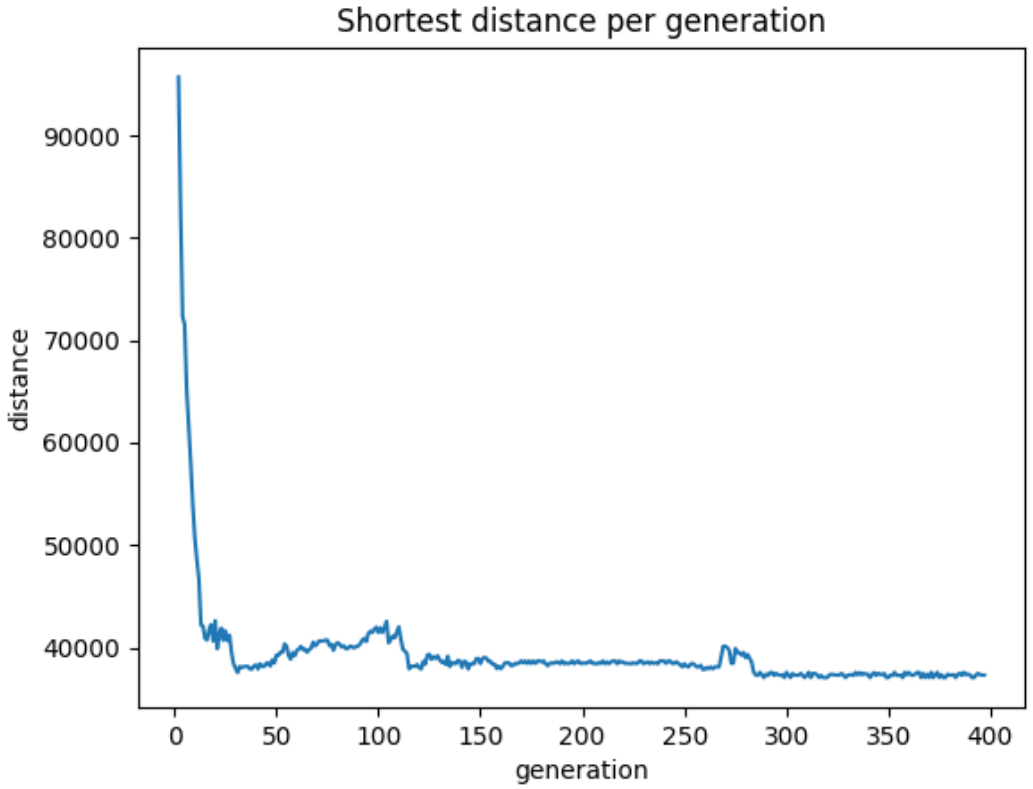

Largest distance per generation
distance
generation
140000
120000
100000
80000
0
100
200
300
400
500

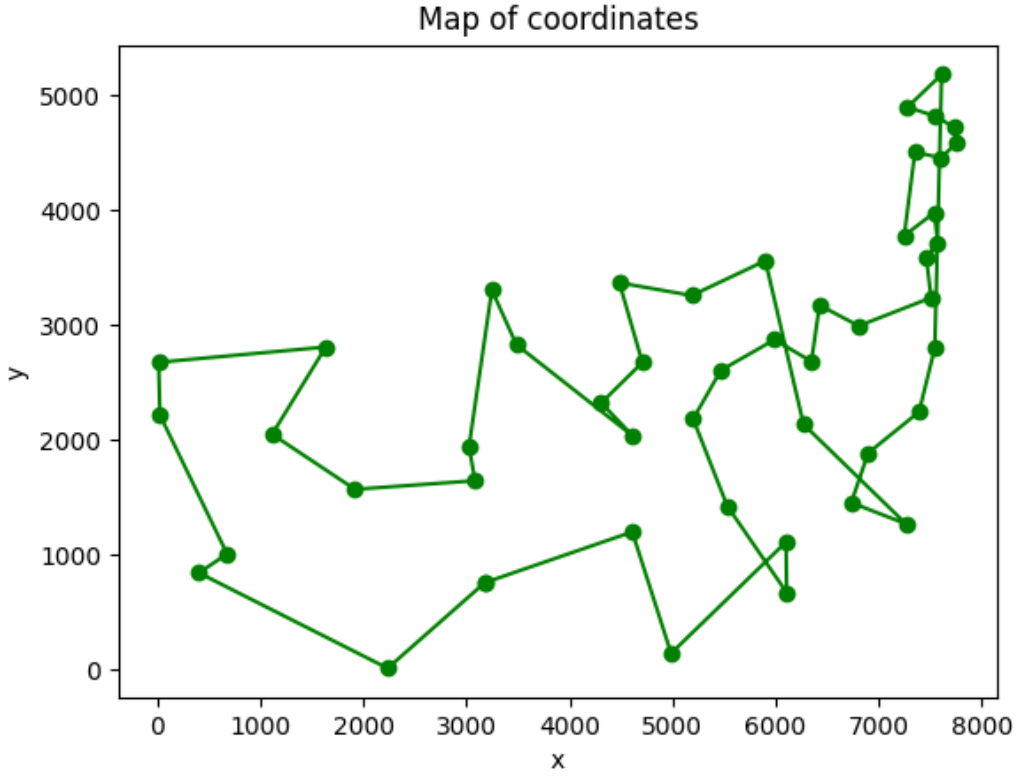

Average distance per generation
distance
generation
120000
110000
100000
90000
80000
70000
60000
50000
0
100
200
300
400
500


Map of coordinates
y
x
5000
4000
3000
2000
1000
0
0
1000
2000
3000
4000
5000
6000
7000
8000

En esta ejecución se piden solo dos archivos. Gráfico de los archivos pedidos:

```
lucia@lucia-laptop:~/Documentos/GitHub/TP-2-3/src$ ./p -f cities.txt -g 397 -u 127 -o 14
```



```
lucia@lucia-laptop:~/Documentos/GitHub/TP-2-3/src$ ./p -f cities.txt -g 870 -u 100 -b
```

En esta ejecución se pide imprimir la mejor solución. Por terminal se imprime:

```
<< Best route found >>
After 870 generations, the best distance achieved was: 36548.3
The cities on this route were in the following order:
(7280,4899)
(7555,4819)
(7762,4595)
(7732,4723)
(7608,4458)
(7352,4506)
(7541,3981)
(7573,3716)
(7462,3590)
(7248,3779)
(7509,3239)
(7545,2801)
(7392,2244)
(6807,2993)
(6426,3173)
(6347,2683)
(6271,2135)
(5468,2606)
(5199,2182)
(4612,2035)
(4307,2322)
(4706,2674)
(5185,3258)
(4483,3369)
(3484,2829)
(3245,3305)
(3023,1942)
(3082,1644)
(1916,1569)
(1112,2049)
(1633,2809)
(10,2676)
(23,2216)
(401,841)
(675,1006)
(2233,10)
(3177,756)
(4985,140)
(4608,1198)
(5530,1424)
(6101,1110)
(6107,669)
(7265,1268)
(6734,1453)
(6898,1885)
(5989,2873)
(5900,3561)
(7611,5184)
```

**Bibliografía:**

*Genetic Algorithms*. (2017b, June 29). GeeksforGeeks. Recuperado de: https://www.geeksforgeeks.org/genetic-algorithms/

Garduño Juárez, R. (2018). Algoritmos genéticos. 2022, Noviembre 21, Conogasi. Recuperado de: https://conogasi.org/articulos/algoritmos-geneticos/