Tarea 2: Desarrollar algoritmo genético para el problema del vendedor viajero

Ignacio Haeussler

25 de noviembre de 2018

1. Problema: TSP

El problema a resolver elegido es el problema del vendedor viajero (TSP por sus siglas en inglés), en el que un vendedor debe viajar por n ciudades una única vez, volviendo a la inicial finalmente, recorriendo la menor cantidad de distancia posible. El problema es modelado por n puntos en el plano cartesiano, y los recorridos pueden ser modelados por permutaciones de los índices de las n ciudades.

Se propone resolver el problema mediante un algoritmo genético, en el que los individuos corresponden a distintas permutaciones, inicialmente aleatorias, de los índices de las ciudades, una operaciones de mutación a un intercambio entre 2 índices aleatorios en una permutación, y la reproducción a una aplicación de crossover ordenado entre dos permutaciones. Adicionalmente, la selección de miembros utilizada consistió en seleccionar una fracción con mayor aptitud, por ejemplo la mitad, el tercio, el cuarto o el quinto de la población con mayor aptitud.

Los tests realizados consistieron en generar 100 archivos de 25 pares de números enteros, para luego aplicar algoritmos genéticos con distintos parámetros u otros algoritmos que resolvieran TSP. El primer test obtiene la distancia total promedio para distintos tamaños de población, fijando como valores iniciales las épocas, tasa de mutación y fracción de selección (150, 0.5 y 1/2 respectivamente). El segundo elige la mejor población y la fija junto a los otros parámetros, excepto la fracción de selección. Finalmente, el tercer experimento compara el mejor obtenido del experimento anterior con los algoritmos aproximados Convex Hull y Minimum Spanning Tree.

2. Sistema operativo, lenguaje de programación y librerías utilizadas

Repositorio de github: https://github.com/masotrix/CppGeneticProgramming. El sistema operativo utilizado fue ubuntu 18.04, el lenguaje fue C++14 y para compilar fue creado una archivo Makefile, el cual una vez estando en el directorio raíz del proyecto, puede ser ejecutado utilizando el comando make. Adicionalmente, éste asume un ambiente linux para crear carpetas asociadas a la creación de los ejecutables.

Por otro lado, la única librería utilizada fue OpenCV 3.4 para realizar gráficos en C++. Para su instalación en Ubuntu 18.04 las instrucciones en el siguiente link fueron seguidas: https://www.pyimagesearch.com/2018/05/28/ubuntu-18-04-how-to-install-opencv/, las que además permiten en ubuntu 18.04 compilar código C++14.

3. Implementación

Una vez estando en la raíz del código y habiendo compilado satisfactoriamente, el siguiente comando ejecuta el programa de la tarea, el cual consiste en el despliegue de gráficos que entregan los resultados asociados a un experimento determinado por distintos parámetros.

```
./build/Arrays/task2
```

Los programas de la tarea están en los archivos src/Arrays/testsTask2 y src/Arrays/task2. Los parámetros mencionados se encuentran al inicio del archivo src/Arrays/task2. El primero es cities, el que determina la cantidad de nodos utilizados en TSP, mientras que los siguientes (members, popDivision, epochs y mutRate) especifican los hiperparámetros utilizados en el algoritmo genético (población, división de la población para seleccionar individuos más aptos, épocas y tasa de mutación, respectivamente). Estos parámetros pueden ser modificados y luego de ejecutar make en la raiz del código, volver a ejecutar el experimento con los parámetros modificados. Adicionalmente, el path de un archivo desde la raiz del código puede ser entregado a ./build/Arrays/task2 para utilizarlo. Por defecto, el programa generará un nuevo archivo con la cantidad de nodos especificada, el que utilizará para todo el experimento.

Adicionalmente, el siguiente comando ejecuta el programa de experimentos, el cual consiste en el despliegue de gráficos que entregan los resultados asociados a los distintos experimentos mencionados en la sección 1:

```
./build/Arrays/testsTask2
```

Por otro lado, la implementación de los algoritmos genéticos están en los archivos src/GP.h, src/GP.cpp y src/GP/Arrays/testArrays.cpp. El ejecutable asociado a los test estaría en build/GP/Arrays/testArrays. La estrategia utilizada para la implementación consistió en una clase genérica 'GeneticAlgorithm', la que implementa el patrón de diseño 'Template', por medio del método 'evolve()'. Este método iterativamente evalúa y reproduce a los individuos, donde la evaluación (método evaluate()) y reproducción (reproduce()) son implementados por medio de métodos de clases hijas de 'GeneticAlgorithm' (son métodos abstractos). Adicionalmente, se espera que el método select(), el que se encarga de seleccionar a los individuos más aptos para ser reproducidos, sea utilizado por el método reproduce().

En este contexto, la clase 'FindTravelerGA' en primera instancia (en su constructor) recibe un archivo donde se encontrarian los nodos, generaría una población de permutaciones aleatorias de los índices de estos nodos y luego evoluciona sucesivamente esta población mediante su método heredado evolve(). Para ello implementa los métodos evaluate(), select() y reproduce(), con evaluate() midiendo la distancia total asociada a cada permutación, select() dividiendo a la población de permutaciones en más y menos aptas (fracción de selección previamente discutida), y reproduce() aplicando crossover ordenado a pares de permutaciones elegidas aleatoriamente dentro de la fracción de selección y luego aplicando mutaciones aleatorias (intercambios entre índices como se explicó previamente) con la probabilidad especificada, al hijo obtenido de la reproducción.

Por otro lado, el archivo que implementa los experimentos para encontrar secuencias de bits y de carácteres está en src/Arrays/testArrays.cpp, y su ejecutable en build/Arrays/testArrays. Este entrega la cantidad de iteraciones requeridas para encontrar las secuencias y los hiperparámetros utilizados. Las implementaciones de cada algoritmo son similares a la de FindTravelGA, pero sus métodos evaluate(), select() y reproduce() son las explicadas en cátedra, es decir, evaluate() mide la cantidad de coincidencias con el arreglo a encontrar de cada secuencia en la población, select() utiliza el algoritmo de torneo con una cantidad especificada de rondas, y reproduce eligue un índice

aleatorio que especificará cuantos elementos serán utilizados de cada padre (single-point crossover). Para FindCharArrayGA se utilizaron las letras de 'a' a la 'z'.

4. Evaluación

Para empezar, se obtuvieron resultados satisfactorios para los problemas de encontrar secuencias de bits y de letras. Medianta el ejecutable build/Arrays/testArrays se pueden realizar los experimentos que se deseen. Estos muestran que el encontrar secuencias puede requerir hasta 2 ordenes de magnitud más de épocas para tamaños similares. Por ejemplo, para una población de 30, un tamaño de 20 para los arreglos, 10 rondas de torneo y 0.2 en tasa de mutación, encontrar arreglos de bits requierió 20 épocas, mientras que encontrar arreglo de carácteres requirió 6695.

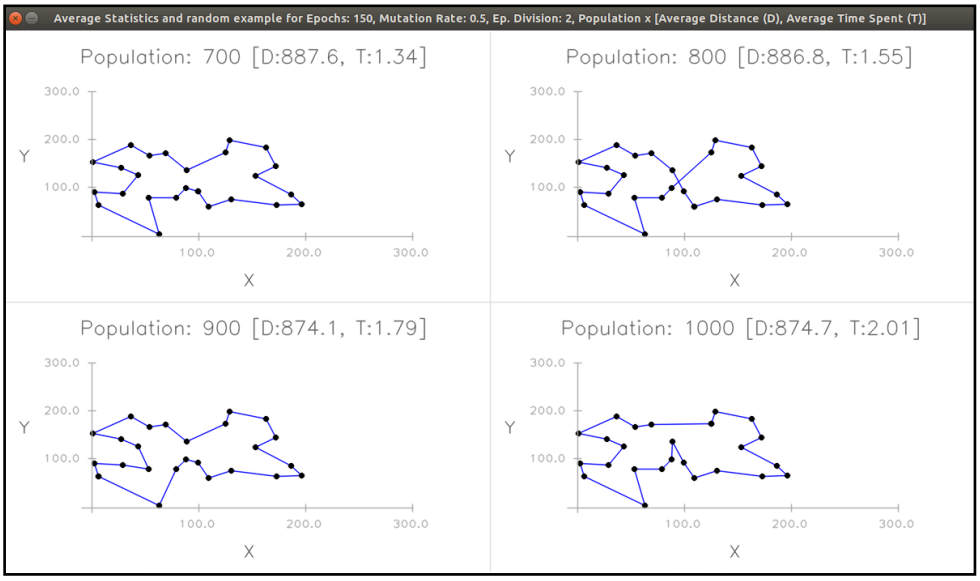

Figura 1: Distancia total promedio, tiempo promedio y ejemplo aleatorio de miembro más apto de algoritmo genético, para 150 épocas, tasa de mutación 0.5 y selección de mitad más apta, para distintos tamaños de población

Se puede apreciar en la Figura 1 que para 900 miembros, el algoritmo obtiene el mejor resultado promedio para los experimentos realizados. La tendencia parece indicar que a una mayor cantidad de población, mejores resultados promedio tienden a obtenerse, con un tope máximo a esta mejora, a la vez que un aumento en el tiempo requerido promedio.

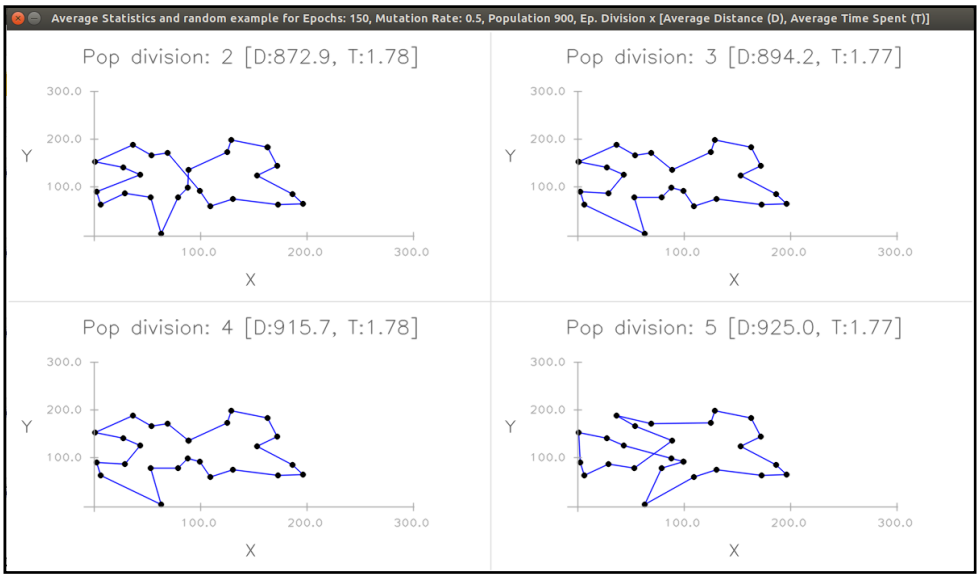

Figura 2: Distancia total promedio, tiempo promedio y ejemplo aleatorio de miembro más apto de algoritmo genético, para 150 épocas, tasa de mutación 0.5 y población de 900 individuos, para distintos niveles de selección (2:mitad más apta, 3:tercio más apto, 4:cuarto más apto y 5:quinto más apto).

En la Figura 2 puede observarse que seleccionar la mitad con mayor aptitud tiende a ser mejor que seleccionar fracciones más pequeñas. Esto puede estar asociado a una disminución perjudicial inicial de la variabilidad de las soluciones encontradas.

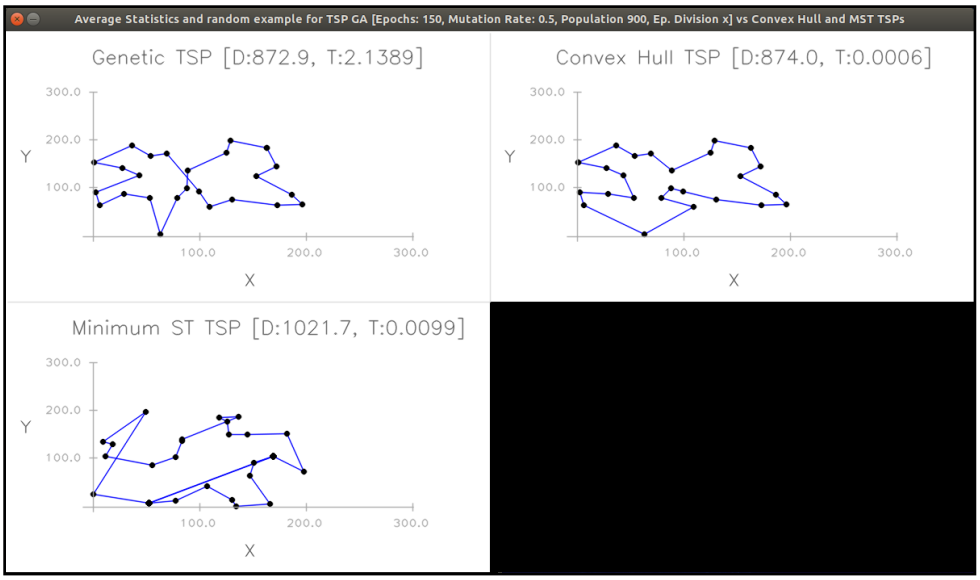

Figura 3: Distancia total promedio, tiempo promedio y ejemplo aleatorio de miembro más apto de algoritmo genético, para 150 épocas, tasa de mutación 0.5, nivel de selección 2 y población de 900 individuos, comparado con algoritmos Convex Hull y Minimum Spanning Tree

Luebo, en la figura 3 se observa que el algoritmo Genético obtenido supera en promedio a los otros algoritmos probados (Convex Hull y Minumum Spanning Tree), a pesar de ser considerablemente más lento. Esto insinúa la hipótesis de que en general los algoritmos genéticos serían inferiores a algoritmos que incorporen una mayor parte de la estructura del dominio del problema que se esté enfrentando.

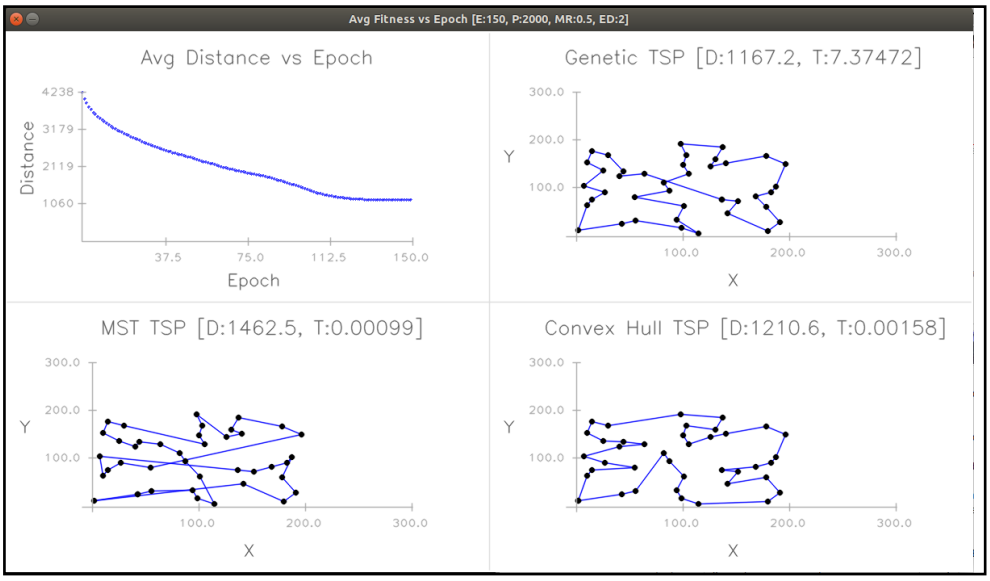

Figura 4: Gráfico de evolución de distancia promedio de individuos, individuo final más apto y resultados para MST y Convex Hull para el mismo gráfico, para 150 Épocas, 2000 individuos, 0.5 de tasa de mutación y selección de la mitad de la población por época.

Finalmente, en la Figura 4 puede observarse el resultado de realizar un experimento sobre un mapa de 200 ciudades. Puede observarse nuevamente la superioridad del algoritmo genético en términos de la distancia total final obtenida, y a la vez su inferioridad en costo temporal. También se presenta la evolución de la distancia total promedio en la población del algoritmo genético a través de las épocas.

5. Discusión

Como puede observarse del resultado de los experimentos, el problema logró ser resuelto al obtener soluciones mejores a algoritmos conocidos. Sin embargo, requiere una cantidad de tiempo mucho mayor a estos. Como ya se mencionó, esto probablemente se deba a que aquellos algoritmos incorporan mayor parte de la estructura del problema, permitiendoles abordarlo desde una perspectiva menos general y signicativamente menos compleja, por lo tanto requiriendo mucho menor trabajo para lograr avances en el descubrimiento de buenas soluciones.

Por otro lado, como se dedujo en el análisis parcial de los experimentos, una selección demasiado elitista es perjudicial para el problema del TSP, de manera que la selección de la mitad más apta de toda la población supera a la selección de una fracción más reducida. Adicionalmente, una mayor

población tiende a ser mejor que una inferior hasta cierto punto. La elección de lo suficientemente grande permitió encontrar permutaciones que superaran a MST y Convex Hull, mientras que al reducir la población se obtienen rápidamente permutaciones finales inferiores.

Finalmente, más experimentos asociados a la influencia de las mutaciones y a la fracción de selección serían requeridos para comprender mejor su influencia en este problema.