UNIVERSIDAD ANDRÉS BELLO

FACULTAD DE INGENIERÍA

INGENIERÍA CIVIL INFORMÁTICA

Trabajo 5 – Problem Solving

ANDRÉS EDUARDO VALENZUELA GONZÁLEZ

SANTIAGO - CHILE

NOVIEMBRE, 2017

Contenido

1. Introducción

El problema del vendedor viajero responde la pregunta: “dada una lista de ciudades conexas entre si y cada conexión posee un coste de viaje, ¿Cuál es la mejor ruta que conecte a todas las ciudades y represente el menor coste de viaje volviendo así a la ciudad de origen?”. Dentro de la optimización combinatoria, este es un problema *NP-duro* (un problema del cual se provee una solución en tiempo polinómico).

Dada la presente problemática, se pueden proponer como solución diversas heurísticas (o bien diseñar alguna) tales como “Algoritmo del vecino más próximo”, “Intercambio par a par”, “Heurística de k-opt o heurística de *Lin-Kernighan*”, “*Bee Colony Optimization* (BCO)”, “*Ant Colony Optimization* (ACO)” o “Algoritmos genéticos”.

Para un algoritmo genético existen diversas implementaciones (seguir un modelo solución, ACO, BCO o diseñar uno propio), los cuales buscan recrear la sabiduría de la naturaleza en cuanto a creación de población, selección de población, evolución y/o mutación para luego repetir el proceso hasta encontrar la mejor solución. Dadas estas características, el algoritmo genético no siempre encontrara la solución óptima pero si una *óptima local*. Como fue mencionado antes, esto dependerá netamente de la implementación y lógica que se le dé al algoritmo.

2. Desarrollo del problema

Se debe implementar uno o más algoritmos para poder realizar problemas TSP de la biblioteca de problemas de la Universidad *Heidelberg*, que entrega un archivo .tsp con la información de los nodos y los caminos:

http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/

Para este informe se solicita probar varias instancias de distintos tamaños y analizar los resultados en términos de calidad de la solución versus tiempo de ejecución. También entre números de iteraciones y tiempo de ejecución, probar distintos parámetros de la estrategia seleccionada de manera que optimice su desempeño, analizar las mejores soluciones encontradas a la luz de las mejores provistas por la biblioteca y por último, proponer mejoras que se podrían hacer para mejorar el desempeño.

3. Analisis de Resultados

Implementación:

Para el desarrollo de este informe el alumnado se vio en la libertad de diseñar su propio algoritmo genético (del caso contrario, comparar los algoritmos ACS y BCO).

En este caso, se propuso un algoritmo genético desarrollado en Python 2.X, orientado a objetos y multi-thread. La implementación cuenta con 3 archivos de los cuales en el archivo *Genetic.py* y *TSP.py* se pueden ubicar 5 clases distintas, las cuales proveen de bastante lógica aplicada para luego solo aplicar multi-threads en el archivo main.py para aprovechar la potencia de los núcleos y así obtener más de un resultado por ejecución.

TSP.py clases:

- *Node*: Clase para representar los ejes X e Y de un nodo dado un identificador. Además posee una única función la cual retorna las coordenadas del mismo nodo como tupla.
- *TSP*: Clase para obtener y parsear los archivos tsp. Los únicos archivos capaces de ser parseados son los *EUC_2D* y *ATT.*

Genetic.py clases:

- *Gen*: Hereda de la clase Node. Un gen es utilizado básicamente para representar un nodo obtenido desde el archivo tsp para luego conformar un cromosoma.
- *Chromosome*: Un cromosoma es la representación de una posible solución, la cual se conforma por una lista de genes. Cabe destacar que la cabeza debe ser idéntica a la cola. Esto representa el inicio y el fin del viaje del vendedor viajero.
  Cada cromosoma tiene una lista de genes, un *fitness* esperado, una función de distancia y una probabilidad de mutación. Tanto el *fitness* esperado como la función de distancia es la misma para todos los cromosomas, en cambio la lista de genes y la probabilidad de mutación son distintos.
  Mencionada clase posee distintos métodos y funciones:
  - *mutation*: función que opera con base a la probabilidad de mutación. Si esta probabilidad es menor a un número *pseudo-random*, el gen puede mutar. Del caso contrario, no puede.
    Mencionada función divide al cromosoma en dos, le aplica una función reverse a cada split y luego los une para generar un nuevo camino.
  - *crossover*: Genera un cruzamiento *random* entre dos cromosomas para crear una nueva posible solución (nuevo cromosoma).
    Dada esta función, si los padres son idénticos, el nuevo hijo no tendrá variabilidad entre sus genes ya que la función toma la cabeza del padre o de la madre (50% de probabilidad de tomar una o la otra) y luego recorre el cuerpo de ambos tomando un gen para incluirlo dentro del cuerpo del nuevo cromosoma.
  - Funciones de distancia: se poseen distintas funciones de distancia (euclidiana, manhattan, *canberra*, *squared cord*, *squared_chi_squared*), de las cuales la euclidiana es la función por defecto para calcular el *fitness* de cada cromosoma.

- *GeneticAlgorithm*: Es la clase encargada de crear, contener las poblaciones, iterar con base a la cantidad de generaciones dadas por parámetro y de seleccionar la población respecto al *fitness* mínimo aceptable. Esta clase posee 3 métodos los cuales son fundamentales para el proceso de iteración del algoritmo genético, los cuales son:
    - *generate_chromosomes*: utilizando los nodos rescatados del archivo tsp, genera los genes del cromosoma y luego un cromosoma completo haciendo uso de la clase *Chromosome* y *Gen*.
    - *generate_population*: Dado un fit entregado por parámetros, se encarga de hacer la llamada a la función *generate_chromosomes* para luego anexar el cromosoma creado a una lista denominada "población", la cual es un conjunto de cromosomas.
    - *run*: Esta función se compone de distintas sub-funciones implícitas, tales como seleccionar la población, realizar el crossover entre otros cromosomas y ejecutar la función de (posible) mutación para cada cromosoma.

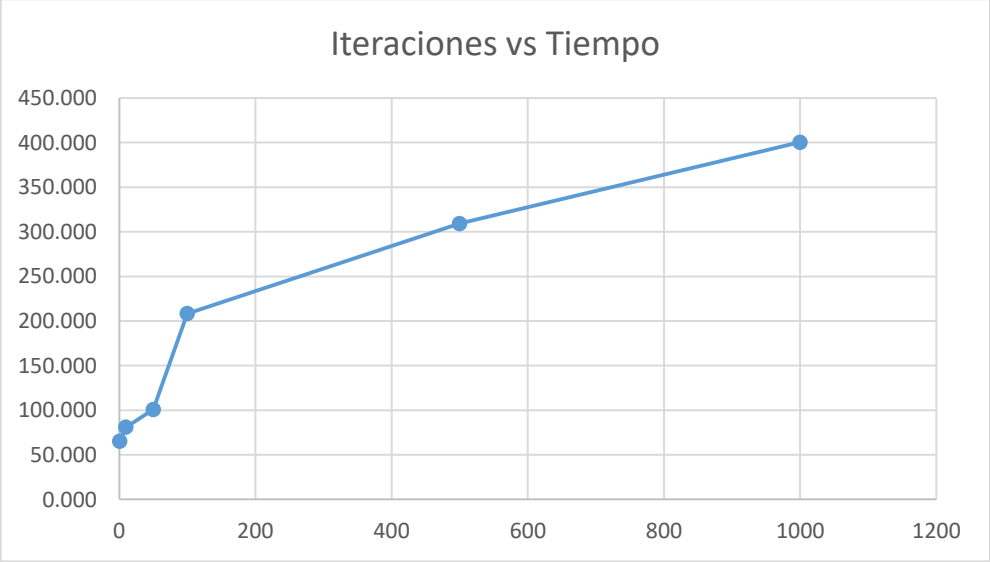
main.py: En este archivo se puede ubicar la lógica necesaria para poder hacer funcionar todo de tal manera que represente un real algoritmo genético (ejecutamos las instrucciones secuencialmente para luego obtener un resultado final).
Se pueden ubicar dos funciones similares, las cuales corresponden al procedimiento para ser llevado a cabo dentro de los hilos que luego se crearan.
Cada función crea una instancia de la clase *GeneticAlgorithm* con los parámetros correspondientes, comienza un ciclo iterativo con base a la generación máxima para esta instancia y luego hace llamado a la función run reduciendo secuencialmente el *fitness*. Esta reducción de *fitness* depende de lo que uno requiera. Para este caso se estimó conveniente ir reduciendo el *fitness* en 40 puntos por cada iteración para acotar la selección de población e ir reduciendo la población apta para la reproducción.
Luego de finalizar el ciclo iterativo con base a la máxima cantidad de generaciones, se procede a registrar al mejor cromosoma con todos sus datos (en formato *json*) para su posterior análisis.
Para el cuerpo del *main*, la línea 70 hace lectura del archivo tsp y genera un objeto TSP, las líneas 79 y 82 instancian los hilos para la próxima ejecución, líneas 80 y 83 setean ambos hilos como "*daemon*" para que no interfieran con la ejecución del hilo principal (más bien para que no tomen el control de *main*) y luego en las líneas 85 y 86 se inicia el proceso de ambos hilos. (No se entrara en detalle de la creación de los hilos ya que no corresponde a esta materia. Solo se comentara que las funciones antes explicadas son pasadas como parámetros a estos hilos para ser ejecutadas).
Finalmente en la línea 88 se mantiene ocupado al hilo principal para que los hilos anteriores no mueran y sigan ejecutándose.

Cantidad de iteraciones vs tiempo:

Para la cantidad de iteraciones se estimó conveniente asignar una por solicitud de usuario para luego intentar resolver el problema att48.tsp y **para la cantidad de población se asignó 1000 por ejecución** (se discutirá sobre esto a continuación). A continuación podemos ver los resultados:

Iteraciones	Tiempo (s)	Tour Óptimo
1	65,292	111.432
10	80,911	109.645
50	100,784	108.894
100	208,276	103.735
500	309,241	104.843
1000	400,504	106.327

Como podemos ver, el mejor tour generado fue con 100 iteraciones y con un tiempo de ejecución de 208 segundos. Se puede contemplar que el tiempo aumenta a medida que el número de iteraciones crece, esto se puede apreciar mejor en el siguiente gráfico:



Variación de parámetros

El primero de los parámetros que se analizo fue el tamaño de la población.

Se dedujo que entre menor cantidad de población, menor será el tiempo de ejecución ya que habrá un espectro menor para ser analizado, pero esto conlleva una gran responsabilidad ya que al reducir la muestra, se reduce la cantidad de posibles caminos óptimos y la cantidad de cruces entre cromosomas (en síntesis, se reduce todo, incluso la posibilidad de encontrar un mejor camino).

En cambio, si este parámetro es muy alto, el algoritmo tomara aún más tiempo en resolver la iteración, pero la calidad de salida que entregue será mayor ya que habrán mas muestras existentes y existirá una mayor probabilidad de encontrar el camino óptimo.

También juega un papel de suma importancia la probabilidad de mutación ya que da a lugar la variabilidad de cromosomas. Con un nivel bajo, los cromosomas tenderán a ser siempre los mismos y habrá poca variabilidad y por ende, menor probabilidad de encontrar un camino óptimo. En cambio, si se posee una probabilidad de mutación moderada, existirá un cambio persistente dentro de los cromosomas, permitiendo así variabilidad en sus soluciones parciales.

Solución obtenida vs solución optimizada

En la página de soluciones, el archivo att48.tsp presenta una solución óptima de 10628 unidades de medida. El mejor resultado se obtuvo con 100 iteraciones y con 1000 cromosomas en un tiempo de casi 4 minutos con un valor de 103.735. Se puede rescatar de esto que el código creado se encuentra muy lejos de lo óptimo y ha caído en un óptimo local del cual no podrá salir hasta que esta solución mute o nazca algún hijo mejor de esta solución y la que la antecede.

Mejorar la solución

Para mejorar este algoritmo genético se pueden aplicar 2/3-opt a la solución final, agregar otros conceptos al algoritmo genético como elitismo, tener una estrategia inicial para la población inicial (otra que no sea aleatoria) y en conjunto, mejorar la implementación del actual algoritmo genético para encontrar una solución en el menor tiempo posible.

4. Conclusiones

Luego del desarrollo de un propio algoritmo genético se concluye que la cantidad de generaciones por ejecución aumenta exponencialmente el tiempo de ejecución del programa, pero a cambio se espera una mejor solución (dependiendo de cómo este desarrollado el algoritmo y que tanto reajuste obtenga el *fitness*).

Una observación que se debe tener en cuenta es el desarrollo del algoritmo genético, es decir, se pueden sobre ajustar a las exigencias que uno encuentre pertinentes ya sean estas el reajuste del *fitness* por cada iteración, la creación de la población sin repetición, el crossover, la mutación, etc. Es decir, todos estos factores y más pueden afectar en gran medida el resultado del algoritmo genético.

También es importante destacar el *fitness* de corte para seleccionar la población, es decir, si es muy alto, toda la población puede sobrevivir, pero si es muy bajo, ningún cromosoma sobrevivirá, lo cual conlleva a un problema al momento de probar estos algoritmos.

Con respecto a la variación de resultados, estos se deben a los puntos anteriores más la aleatoriedad con la que se genera la población ya que no siempre serán las mismas poblaciones por cada ejecución.

En cuanto al tiempo, cabe considerar que el algoritmo genético fue desarrollado en un lenguaje interpretado, por lo que su tiempo de ejecución se puede ver mermado ante tantas instrucciones y tantos ciclos iterativos (esto último puede considerarse para luego mejorar el tiempo de ejecución implementando algoritmos de búsqueda más eficientes o implementar otras funciones de reproducción o mutación, etc.). Con respecto a la variación de parámetros, esto último marca también una diferencia significativa en cuanto a cantidad de generaciones, iteraciones, probabilidad de mutación entre otros (respecto al tiempo de ejecución)

Con respecto a la variación de parámetros, esto marca una gran diferencia en cuanto a cantidad de generaciones, iteraciones, probabilidad de mutación entre otros (respecto al tiempo de ejecución)

Además, la solución propuesta por la biblioteca de soluciones era muchísimo menor que las soluciones optimas locales encontradas por el algoritmo actual desarrollado. Esto pudo deberse a distintas implementaciones, optimizaciones, uso de distintas heurísticas entre otras.