PROYECTO 1: TSP Recocido Simulado con aceptación por umbrales

Seminario de Heurísticas de Optimización Combinatoria

Alumna:

Karla Adriana Esquivel Guzmán

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

11/Noviembre/2018

Como primer proyecto del curso se nos asignó la tarea de resolver TSP usando recocido y simulado con aceptacion por umbrales, los algoritmos que se usaron nos fueron proporcionados por el profesor en 2 archivos en formato PDF. Por lo que la principal tarea del proyecto era hacer la correcta implementación de los algoritmos.

- **SQLite**
  Los datos de las ciudades nos fueron proporcinadas dentro de un archivo para ser cargadas en una base de datos. Para esto elegí SQLite3 por su sencillo uso y que la mayoría de los lenguajes de programacion tienen bibliotecas para utilizarlas.

- **Lenguaje de programación**
  Elegí el lenguaje de progrmacion GO porque nunca habia pogramado nada en el y me pareció interesante por su parentesco a C.
  Para la instalción del lenguaje me referí a la pagina oficial del lenguaje (https://golang.org/) y segui las instruccion ahí descritas, como utilicé SQLite necesitaba dicha biblioteca. GO lo instalé siguiendo las instrucciones que están en el respositorio del proyecto (https://github.com/mattn/go-sqlite3).

- **GNUPlot**
  Para hacer las gráficas para mostrar las soluciones aceptadas por el algoritmo use GNUPlot.

- **Implementación**
  La implementación no es la más elegante que hay, por que toda fue hecha dentro de un mismo archivo tsp.go, mi plan era primero terminar la implementación del problema ya teniendo soluciones óptimas reestructurar el código pero para esto último ya no me dio tiempo.

- **Structs**

```
type city struct {
   Id int
   Name string
   Country string
   Population int
   Latitude float64
   Longitude float64
}
```

Esta estructura es para cargar toda la información referente a una ciudad y poderlo manipular fácilmente.

```
type temp_parametes struct {
   teInicial float64
   pmayus  float64
   epsilomTe float64
   epsilomPe float64
   ene int64
}
```

Esta estructura contiene los parámetros necesarios para calcular la temperatura inicial.

```
type tsp_parameters struct {
   maximaDistancia float64
   normalizador float64
   citiesDistance map[int]map[int]float64
   tamLote float64
   intentosMaximos float64
   factorEnfriamiento float64
   apuEpsilon float64
}
```

Esta estructura contiene los parámetros necesarios para calcular la temperatura inicial, así como valores que necesitamos conocer en varias etapas del algoritmo como, la distancia Máxima entre las ciudades, el valor del normalizador y un diccionario (map en GO) dónde está las distancias entre todas las ciudades que se utilizan para la instancia del problema.

- **Funciones**

`listaCiudades(entradaSize int, ciudadesIds string) []city`

Dada la lista de ciudades como una cadena de los ids separada por comas y la cantidad de ciudades, obtenemos de la base datos la información de estas ciudades y los metemos un arreglo de la estructura city.

`printListaCiudades(cities []city)`

Dado un arreglo de city imprimo la información de las ciudades en dicho arreglo. Esta función la utilicé principalmente para hacer debug.

`listaDistancias(cities []city) []float64`

Dado un arreglo de city obtenemos de la base de datos la distancia entre todas las ciudades y lo regresamos en un arreglo de float64 ordenado.

`listaNormalizador(distancias []float64, entradaSize int) []float64`

Dado un arreglo de las distancias entre las ciudades y la cantidad de ciudades N que nos dan como entrada, obtenemos las N mayores distancias y las regresamos una lista, esta lista es usada para calcular el normalizador.

`getNormalizador(listaNormaliza []float64) float64`

Dada la lista de las mayores distancias entre las ciudades calculamos el valor del normalizador.

`distanciaNatural(ciudad1 city, ciudad2 city) float64`

Calcula la distancia natural entre 2 ciudades usando el algoritmo que nos fue proporcionado en uno de los PDF, este algoritmo no tiene gran complicación ya que dada las coordenadas de las ciudades unicamente se hacen operaciones matemáticas para obtener la distancia.

`pesoAumentado(ciudad1 city, ciudad2 city, maximaDistancia float64)`

Calcula el peso aumentado entre las 2 ciudades, si esta conexión está definida en la base de datos regresa esa distancia de otra manera, calcula la distancia natural y la multiplica por la maximaDistancia. Este algoritmo tambien se nos dio en uno de los PDFs.

`funcionCosto(cities []city, tspParams tsp_parameters) float64`

Calcula la función costo de la solución, básicamente lo que hace es obtener el peso aumentado entre las ciudades contiguas en el arreglo de ciudades (este orden es la solución) suma estos pesos y el resultado de la suma lo divide entre el normalizador.

`vecino(random *rand.Rand, cities []city) []city`

Dada una lista de city hace una copia de esta lista y hace el swap de 2 ciudades de manera aleatoria en la nueva lista y la regresa.

`porcentajeAceptados(random *rand.Rand, cities []city, te float64, tempParams temp_parametes, tspParams tsp_parameters) float64`

Decimos que una solución Y es aceptada si esta solucion es calculada a partir de aplicarle la función vecino a una solución X y $funcionCosto(Y) < funcionCosto(X) + Temperatura$

Calculamos el porcentaje de soluciones que se "aceptan" con los parámetros dados y una temperatura fija. Esta función es utilizada para calcular la temperatura inicial.

`busquedaBinaria(random *rand.Rand, cities []city, te1 float64, te2 $float64$, tempParams temp_parametes, tspParams tsp_parameters) float64`

Esta funcion usa la funcion porcentajeAceptados para ayudar a calcular la temperatura inicial este algoritmo esta definido en uno de los PDF.

`temperaturaInicial(random *rand$.Rand, cities []city, tempParams temp_parametes, tspParams$ tsp_parameters) float64`

Esta funcion dada una temperatura calcula la temperatura inicial que usaremos para el resto del algoritmo, usando las funciones porcentajeAceptados y busquedaBinaria como esta definido en el PDF.

`calculaLote(random *rand.Rand, temperatura float64, cities []city, bestCities []city, tspParams tsp_parameters)(float64, []city, bool, []city)`

A Partir de una lista city (cities) con la última solución aceptada hasta el momento, una lista city (bestCities) con la mejor solución aceptada hasta el momento y los parámetros tspParams.

Calcula un lote de soluciones como está definido en uno los PDF, con un par de cambios que es que guarda la mejor de las soluciones calculada y si después de una cantidad definida de intentos no puede generar un lote de ciudades se detiene.

Regresa 4 cosas el promedio de las soluciones del lote, una lista de city con la última solución aceptada, un booleano que nos dice si debemos detener el algoritmo y una lista de city con la mejor solución aceptada hasta el momento.

`aceptacionPorUmbrales(random *rand.Rand, temperatura float64, cities []city , tspParams tsp_parameters)`

Este metodo calcula una solucion para el problema de TSP dada una tempera inicial, la lista cities con una solución inicial y los parámetros para el algoritmo tspParms.

Esto lo hace como está definido en el PDF mas un par de cambios para detenernos antes si no podemos seguir mejorando las soluciones encontradas.

`appendFile(file_name string, string_to_write string)`

Esta función simplemente agrega al final del archivo $file_name$, la cadena $string\_to\_write$, esta función la utilize para crear los archivos con los resultados.

`citiesACadenaIndices(cities []city) string`

Convierte una lista de city a la una cadena con los ids de las ciudades, esta funcion la use principalmente para debug.

`printMapCities(citiesDistance map[int]map[int]float64)`

Imprime el diccionario de distancias entre ciudades esta funcion fue usada para debug.

`aleatorizaSolucion(random *rand.Rand, cities []city) []city`

Dada una lista de citys hace una copia de esta lista y hace $2 * tamao$ de lista swaps de ciudades de manera aleatoria y regresa dicha lista.

- **Main()**
  Es la función principal que ejecuta lo necesario para calcular la solución a TSP. Esta función se puede ver como divida en 5 Secciones (Marcadas como comentarios en el código fuente).

  1. **Sección 1: Definimos la instancia de TSP.**
     El tamaño de la entrada, una cadena con los ids de las ciudades a partir de las 2 cosas anteriores y con ayuda del método listaCiudades creamos la lista de las ciudades con toda la información necesaria para calcular TSP.
  2. **Sección 2:**
     Dado el arreglo de las ciudades calculada en la sección calculamos la distancia maxima y el normalizador para esta instancia de TSP.
  3. **Sección 3:**
     Para ahorrarnos tiempo y hacer más eficiente siempre que necesitemos saber la distancia entre 2 ciudades, creamos una estructura que dados los IDs de 2 ciudades se la pasamos esta estructura y nos regresa la distancia.

     Esta estructura es un map que dado un entero nos devuelve otro map, este segundo map cuando le damos un entero nos regresa un float. De tal manera que los 2 enteros que le damos a los 2 niveles de maps son las IDs de las ciudades y los que nos regresa el segundo map es la distancia entre estas 2 ciudades.

     En esta sección creamos esta estructura que usaremos para obtener las distancias entre 2 ciudades siempre que lo necesitamos, de esta manera solo calcularemos 1 vez la distancia entre cualquiera 2 ciudades.
  4. **Sección 4:**
     Definimos las estructuras que tienen los parámetros que usaremos para calcular la temperatura inicial y la aceptacionPorUmbrales.
  5. **Sección 5:**
     En esta sección definimos 2 variables intInicio y intFinal y un for que itera entre los valores [intInicio, intFinal] y utilzia estos valores como semilla para la función random que utilizamos durante la ejecución de los diferentes funciones que hacen uso de un random. Dentro del for que hacemos es calcular valor de la semilla random, primero una temperatura inicial y posteriormente, una

solución al problema TSP usando aceptacionPorUmbrales, vamos guardando los resultados en un archivos "fileResults.txt".

- **Proceso de implementación**

  Durante la implementación de las funciones descrita en el apartado anterior, tuve diferentes problemas debido a que el lenguaje de GO era nuevo para mi y no sabia muchas cosas como por ejemplo como se pasaban los arreglos al llamarlos como parámetro de una función, encontrar y entender cómo funcionaban los maps también fue algo que me costó trabajo, así como entender los "arreglos" o slices que usa GO. Creo que algo que me saboteo bastante y de haberlo hecho antes pude haber encontrado más rápido soluciones fue que:

  1. Para saber si una solución era factible divida el costo de la solución entre la distancia máxima
  2. Para calcular la distancia entre ciudades cuando calcula la distancia de (x,y) y existía en la base de datos la distancia de (y,x) y no la distancia (x,y) yo no ponía la distancia (x,y) = distancia(y,x) si no que calculaba la distanciaNaturalxDistanciaMaxima.

     Y la combinación de estos 2 errores al obtener soluciones factibles se parecían mucho a los valores 0.XX etc que estaba buscando. Entonces estuve mucho tiempo con estos 2 errores que me daban resultados completamente engañosos y modifica cosas que no me modificaban demasiado los resultados. Lo más triste es que el problema (2) fue lo último que resolví y estos es algo que obviamente modifica todos los cálculos que hacía que la distancia entre las ciudades es la base para hacer todos los cálculos.

- **Resultados**

  Obtuve resultados exitosos, dichos resultados están guardados en los archivos Results40.txt y Results150.txt que son los resultados obtenidos con las semillas 0-10 con las instancias de 40 y 150 ciudades respectivamente.

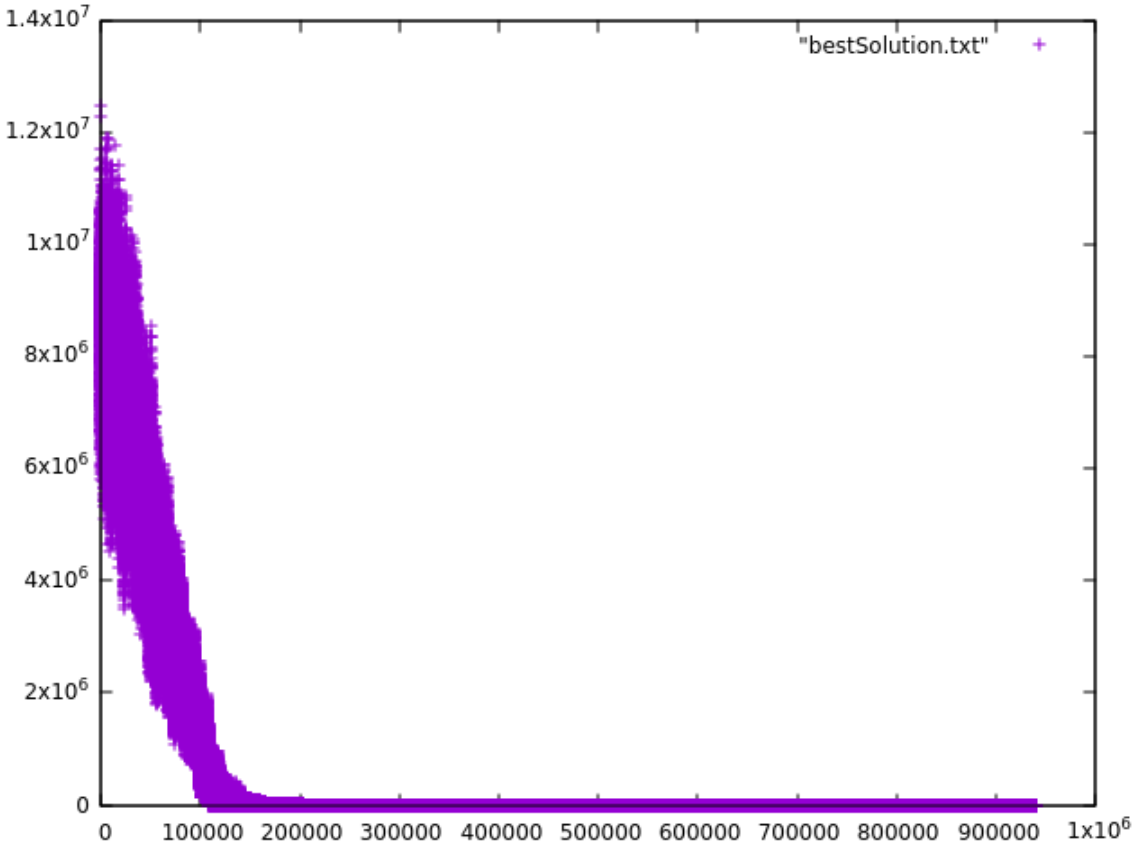
  Algo que me parece curioso es que (aunque no sorprendente debido a cómo funciona el algoritmo) la selección de la semilla es algo que define un poco qué tan bien o mal funciona el algoritmo sin cambiar ningun otro parametro.

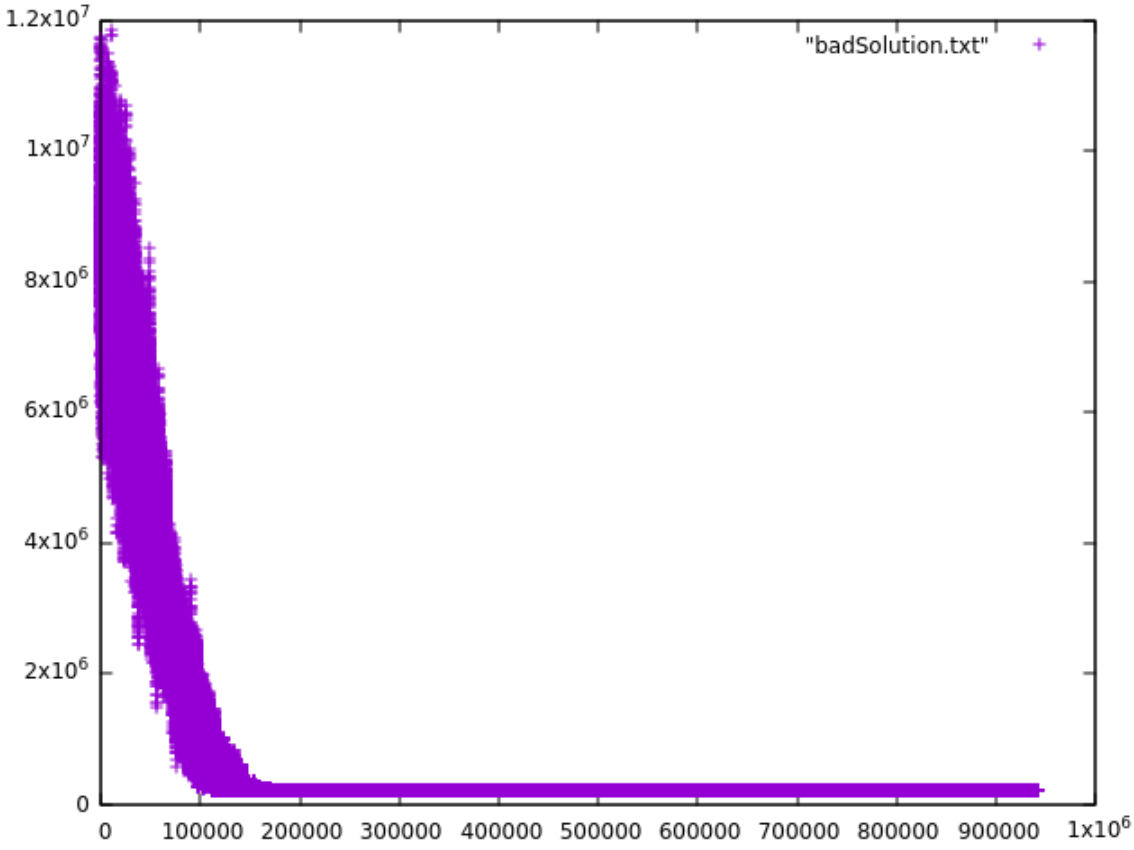
  Como se ve en la figura1 que grafica todas las soluciones aceptadas cuando se encontro una solucion con factibilidad 0.33XXX y la figura2

que encontre una solucion con factibilidad 205601.33XXX

Las graficas se ven muy similares pero simplemente en la figura1 en algún momento se tuvo la suerte de llegar a una solución a partir de la cual se pudieron ir generando soluciones mejores y en la figura2 se quedó "atorado" en soluciones malas, y al ir cambiando la temperatura ya no se le permitió "salir" del hoyo de soluciones malas.

Pero en general las gráficas de las soluciones obtenidas en los 2 casos nos muestra cómo el comportamiento es similar pero simplemente en uno se tuvo "suerte" y en el otro no.




"badSolution.txt"
1.2x10^7
1x10^7
8x10^6
6x10^6
4x10^6
2x10^6
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6