Facultad de Ciencias, UNAM
Resolviendo el problema del agente viajero usando recocido simulado

Marisol Amézcua Lopez

1. Introducción

El problema del agente viajero ha sido muy conocido en la historia de las Matemáticas y Ciencias de la Computación, se sabe que no es posible resolverlo usando un algoritmo determinista en un tiempo acotado por un polinomio, por lo que se procede a intentar resolverlo o aproximarse a una solución usando otros métodos. En este escrito se describirá una forma de realizar esto usando recocido simulado usando aceptación por umbrales.

2. El problema

El problema puede ser resumido como: Si un agente viajero quiere visitar exactamente cada ciudad de una lista de m ciudades (donde el costo de visitar desde la ciudad i a la ciudad j es de c_{ij}) y después de esto regresar a la ciudad de origen, la pregunta entonces es: ¿Cuál es la ruta que el viajero puede tomar para que el costo sea el mínimo? [1].

La importancia de este problema es que es un representante de una clase de problemas conocidos como problemas de optimización combinatoria, la cual a su vez se encuentra en una clase que en Complejidad Computacional se conoce como problemas NP-Duro. A la fecha no se ha encontrado un algoritmo que en tiempo polinomial pueda resolver el problema del agente viajero.

2.1. Características

El primer paso es modelar lo anterior de una forma matemática, para esto se modelará el conjunto de ciudades con una gráfica no dirigida, donde cada nodo representará una ciudad y la arista que las conecta contendrá el costo de viajar de la ciudad i a la j.

En este trabajo no se contemplara el viaje redondo, es decir, nos abstendremos te tomar una última arista que va desde la ciudad final a la ciudad inicial, por lo que nuestra ruta se puede ver como un camino Hamiltoniano. El costo final será la suma de los costos de cada arista usada para la ruta.

Para la implementación de el sistema en este artículo se usará una base de datos de ciudades la cual cuenta con diversas ciudades, sus coordenadas geográficas y conexiones entre las mismas, como

aclaración no se cuenta con todas las posibles conexiones para cada ciudad, es decir, la base de datos no nos modela una gráfica completa. Para corregir esto en la implementación se procede a agregar nuevas aristas con pesos mucho mas grandes que cualquier arista. Para esto hacemos lo siguiente:

Sea $G(V, E)$ una gráfica no dirigida y $S \in V$ una instancia de TSP con la que trabajaremos (ciudades que entraran en el problema).

1. Primero obtenemos la distancia máxima de S, denotada por $max_d(S)$ como:

$$max_d(S) = max(\{w(u, v)|u, v \in S \wedge (u, v) \in E\}) \tag{1}$$

2. Obtenemos la distancia natural para cada par de vértices tales que su arista no esta en la base de datos. Para esto definimos la distancia natural $d(u, v)$ como:

$$d(u, v) = R * C \tag{2}$$

Donde R es el radio del planeta Tierra en metros (aproximadamente 6,373,000) y C está dada por:

$$C = 2 * arctan(\sqrt{A}, \sqrt{1 - A}) \tag{3}$$

y A a su vez es:

$$A = sin\left(\frac{lat(v) - lat(u)}{2}\right)^2 + \left[cos(lat(u)) * cos(lat(v)) * sin\left(\frac{lon(v) - lon(u)}{2}\right)^2\right] \tag{4}$$

donde lat(u) es la latitud de la ciudad u y long(u) es la longitud de u.

3. Al final el peso aumentado será:

$w(u, v) = max_d(S) * d(u, v)$

Para esta implementación utilizaremos un normalizador, el cual será el costo de la ruta mas pesada, para esto ordenamos las aristas que estan en la base de datos de mayor a menor costo y tomamos las $|S| - 1$ aristas mas pesadas, a este conjunto le llamaremos L. El normalizador de la instancia será entonces:

$$\mathcal{N}(S) = \sum_{a \in L} costo(a) \tag{5}$$

Nuestro costo para cierta ruta será:

$$f(S) = \frac{\sum_{i=2}^{k} w_S(v_{i-1}, v_i)}{\mathcal{N}(S)} \tag{6}$$

Esto nos da costos de rutas factibles (rutas que no tienen aristas artificiales) entre 0 y 1.

3. La heurística

Como se dijo en la introducción se utilizará aceptación por umbrales, la cual es una heurística que usa como base el recocido simulado de propuesto por Kirkpatrick. El recocido simulado es un algoritmo de búsqueda el cual se inspira en el enfriamiento del metal cuando se calienta para modificar sus propiedades físicas.

Sea entonces dado un problema P clasificado como NP-duro, S el conjunto de posibles soluciones a una instancia de P. Vamos a suponer que tenemos una función $f : S \to \mathbb{R}^+$ (que llamaremos función objetivo), tal que $0 \leq f(s) < \infty$ para cualquier $s \in S$. Dadas $s, s_0 \in S$, si $f(s) < f(s_0)$, entonces consideraremos a la solución s mejor que a la solución s_0 .

Entonces si, de alguna manera, podemos convertir una solución s en una solución distinta s_0 (y viceversa), consideraremos a s y s_0 como soluciones vecinas.

Con lo anterior claro decimos que la idea central de la aceptación por umbrales es, dada una temperatura inicial $T \in \mathbb{R}^+$ y una solución inicial s (ambas arbitrarias), de forma aleatoria buscar una solución vecina s_0 tal que $f(s_0) \leq f(s) + T$, y entonces actualizar la mejor solucion. Continuamos este proceso mientras la temperatura T es disminuida paulatinamente si cumple una serie de condiciones; el proceso termina cuando $T < \epsilon$ (para $\epsilon > 0$ muy pequeña), cuando hemos generado un determinado número de soluciones aceptadas o cuando otra serie de condiciones es satisfecha.

El algoritmo será explicado a detalle en la sección de implementación.

4. Implementación

Para la implementación se utilizó como lenguaje de programación **Julia** y una base de datos para almacenar ciudades con sus conexiones y pesos. Para modelar la gráfica se utilizó una matriz de adyacencias y como se comentó en la sección 2 creamos nuevas aristas las cuales tendrán un peso mayor que cualquier par encontrado dentro de la base de datos.

Entonces para la grafica hacemos lo siguiente:

```
"#Arguments
- ciudades_del_problema:: Array{Int64,1}: Lista con los id's de
 las ciudades que participan en el TSP"
function crea_matriz_adyacencias(ciudades_del_problema)
        norm = normalizador(ciudades_del_problema)
        entorno = get_tabla_min(ciudades_del_problema)
        m = size(entorno)[1]
        n = size(ciudades_del_problema,1)
        matriz = ones(n,n)
        fill!(matriz,0.1)
        for i in 1:m
```

```
                id1 = find_id(ciudades_del_problema, entorno[1][i])
                id2 = find_id(ciudades_del_problema, entorno[2][i])
                distancia = entorno[3][i]
                matriz[id1, id2] = distancia
                matriz[id2, id1] = distancia
        end
        max = get_max(ciudades_del_problema)
        #Matriz llena con distancias de la bd
        for i in 1:n
                for j in 1:n
                        if matriz[i,j] == 0.1
                                matriz[i,j] =
                                 distancia_natural(ciudades_del_problema[i],
                                 ciudades_del_problema[j])*max
                        end
                        if i == j
                                matriz[i,j] = 0
                        end
                end
        end
        return matriz
end
```

Como se puede ver definimos una matriz donde el M[u,v] tiene costo c_{uv} al igual que M[v,u], si u = v entonces M[u,v]=0 y si la arista no esta en la base de datos calculamos la distancia natural y la multiplicamos por la arista con mayor peso.

Una vez con la matriz definida (que es nuestra gráfica) pasamos a implementar la aceptación por umbrales, para esto primero hay que tener en cuenta que una solucion es vecina de otra si hay una permutación entre ellas, si vemos entonces a las soluciones como arreglos la permutación solo es un cambio de posición para un par de valores del arreglo. Nótese que hacer esto es posible porque porque trabajamos sobre una gráfica completa.

Para la aceptación por umbrales comenzamos calculando la temperatura inicial del algoritmo pasando una temperatura arbitraria, para esto se realizará una búsqueda de la siguiente manera:

```
"#Arguments
- s:: Ruta inicial
- T:: Float64: Temperatura inicial de la configuracion
- P:: Float64: Probabilidad de aceptacion de una solucion"
function  temperatura_inicial(s, T, P)
p = porcentaje_aceptados(s, T)
        if abs(P - p) <= epsilon_p
                return T
        end
        if p < P
                while p < P
```

```
                T = 2*T
                p = porcentaje_aceptados(s, T)
        end
        T_1 = T/2
        T_2 = T
    else
        while p > P
                T = T/2
                p = porcentaje_aceptados(s, T)
        end
        T_1 = T
        T_2 = 2*T
    end
    return busqueda_binaria(s, T_1, T_2, P)
end
```

En el código anterior búsqueda binaria se implementa como tal y *porcentaje_aceptados* es como sigue:

```
"Funcion que calcula el porcentaje de
soluciones aceptadas
#Arguments
- T:: Float64: Temperatura inicial
- S:: Ruta actual"
function porcentaje_aceptados(S,T)
    c = 0
    costo_i = costo(S)
    for i in 1:veces
    #Obtenemos una permutacion
        id_s = sample(1:size(ciudades_del_problema)[1], 2,
         replace=false) #obtenemos 2 id's
        id_1 = id_s[1]
        id_2 = id_s[2]
        s_1 = permuta(copy(S),id_1,id_2)
        costo_aux = costo(s_1)
        if costo_aux < costo_i + T
            c = c+1
            S = permuta(S,id_1,id_2)
        end
    end
    return c/veces
end
```

Esta función como su nombre dice calcula el número de soluciones aceptadas dada la temperatura T actual y una ruta. Una vez con la temperatura inicial usamos valores $epsilon_p$, *epsilon*, *phi*,L e *iteracion_max* (obtenidos por experimentación) para realizar lo que es ya la aceptación por

umbrales como sigue:

```
"Funcion que se encarga del recocido simulado
#Arguments
- T:: Float64: Temperatura inicial
- S:: Ruta inicial"
function aceptacion_por_umbrales(T,S)
        p = 0
        while T > epsilon
                q = Inf
                while p <= q
                        q = p
                        a = calcula_lote(T, S)
                        p = a[1]
                        S = a[2]
                end
                T = phi*T
        end
        return S
end
```

La función anterior se encarga de la aceptación por umbrales, observese que la temperatura va decreciendo por un valor phi al cual se le llama factor de enfriamiento y determina que tan lento o rapido decrece la temperatura; el valor de epsilon determina un cero virtual para que cuando lleguemos a esa temperatura nos detengamos. Ahora, otro proceso importante es $calcula\_lote(T, S)$, el cual es como sigue:

```
"Funcion que calcula el lote (soluciones aceptadas)
#Arguments
- T:: Float64: Temperatura inicial
- S:: Ruta inicial"
function calcula_lote(T, S)
        c = 0
        i = 0
        r = 0.0
        costo_i = costo(S)
        while c < L || i < iter_max
                #Obtenemos una permutacion
                id_s = sample(1:size(ciudades_del_problema)[1], 2,
                 replace=false) #obtenemos 2 id's
                id_1 = id_s[1]
                id_2 = id_s[2]
                s_1 = permuta(copy(S),id_1,id_2)
                costo_aux = costo(s_1)
                if costo_aux < costo_i + T
                        S = permuta(S,id_1,id_2)
                        c = c+1
```

```
                    r = r + costo_aux
                    costo_i = costo_aux
                end
                i += 1
        end
        return [r/L,S]
end
```

Un lote en lo anterior es un número determinado de soluciones L, notese que sin la segunda condición de una iteracion máxima el algoritmo podría no terminar.

5. Experimentación

Se comenzó con un conjunto de 40 ciudades de prueba para ver el funcionamiento del sistema.

Después de varios cambios de variables se llegaron a los siguientes parámetros iniciales para el sistema:

- Temperatura inicial: 50
- Tamaño de lote: 2000
- Iteración máxima: 1600
- ϵ: 0.0001
- ϕ: 0.9
- ϵ_p: 0.04
- P: 0.9
- veces: 100

y se logró un costo final de **0.3083307807989823** con la semilla 9 y la ruta graficada a continuación:

Al graficar los cambios en los costos obtenemos la siguiente gráfica:

Como se puede ver es probable que se pudiera reducir la ϵ, sin embargo al intentar hacer esto usando la misma semilla no se obtiene la mejor solución encontrada.

La temperatura disminuye como sigue:

En la gráfica se puede ver de nuevo que la ϵ se podría disminuir. Dentro de estas pruebas se logro un promedio de soluciones factibles.

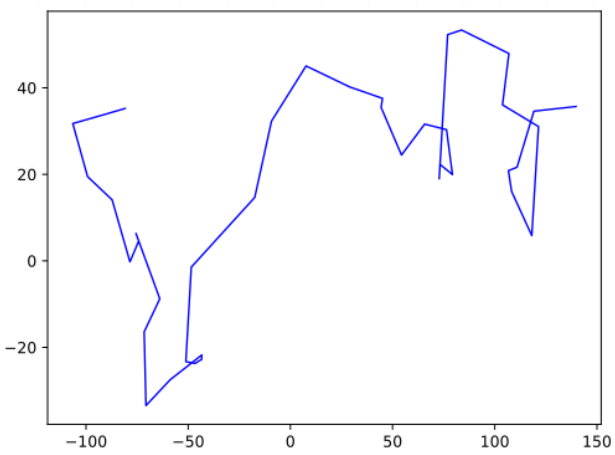

Figura 1: Grafica de la ruta obtenida

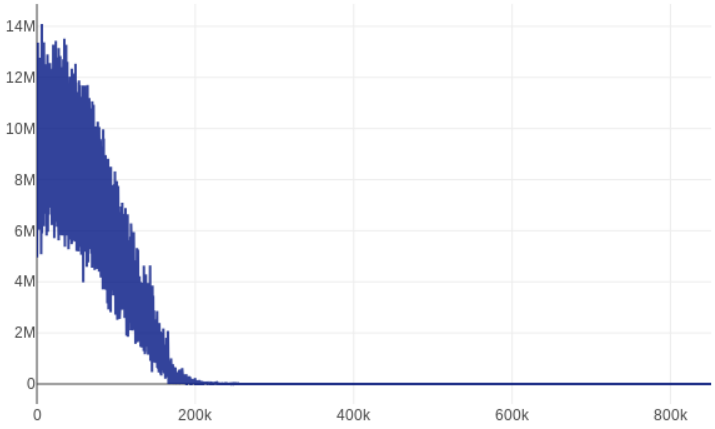

Figura 2: Gráfica de iteraciones contra tiempo

Referencias

[1] Hoffman, K. L. ,Padberg, M. & Rinaldi, G.. (2013). Traveling Salesman Problem. En Encyclopedia of Operations Research and Management Science(pp. 1537-1579). Boston, MA: Springer US.

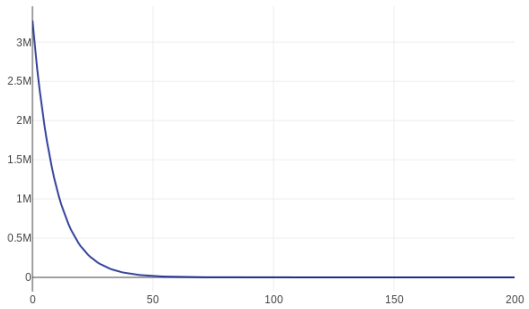

Figura 3: Gráfica de iteración contra temperatura