Universidad Veracruzana

Inteligencia Artificial

Análisis de Algoritmos

Proyecto:
Travelling Salesman Problem

Leonardo Flores Torres

21 de diciembre de 2022

Resolución del problema del viajero utilizando recocido simulado.

A partir de las coordenadas de las capitales de México, en cada ejecución del algoritmo:

1. Dar el valor del número N de ciudades que se utilizaran.
2. Hacer una selección aleatoria de N' ciudades.
3. Resolver el problema del viajero para ese conjunto de ciudades empleando el algoritmo de recocido simulado.
4. Para un número N' de ciudades menor que 10 compare la solución d_s obtenida por recocido simulado con la solución $d^\star$ que se genera al computar todas las posibilidades $N'!$ mediante fuerza bruta. La comparación debe realizarse en términos del error en la distancia, $E = |d^\star - d_s|$.
5. Probar con al menos 3 valores distintos de N, y donde uno de esos valores sea la totalidad de las ciudades capitales de México.
6. Mostrar de manera gráfica la solución final obtenida en cada caso.

Este problema, *the travelling salesman problem*, es uno de encontrar la ruta más óptima ¿Pero óptima en qué sentido? La proposición inicial del problema fue el de encontrar la ruta que minimize la distancia al recorrer un conjunto de puntos de interés, aunque podría extenderse este concepto a encontrar la ruta que minimize el tiempo al recorres estos puntos, o el costo monetario para completar la ruta. Este problema es, en principio, un problema de minimizar una función y la manera de hacerlo es haciendo alusión a un sistema físico.

Primero se podría pensar, y con toda razón, que es un problema de encontrar la combinación del orden en que se visitan estos puntos de interés la cual dé como resultado la distancia más corta de todo el recorrido. Este razonamiento tiene sentido teóricamente pero hay un problema con ello, al aumentar el número N de puntos de interés a visitar el número de combinaciones crece como $N!$. Si en México se quisieran visitar 5 ciudades entonces se tendrían que obtener todas las permutaciones posibles. El número total de permutaciones es igual a

$$ {}_nP_r = \frac{n!}{(n-k)!}, $$

donde n es el tamaño del conjunto, en este caso la cantitad total de puntos de interés, y k es la el tamaño del subconjunto, que corresponde a la cantidad de puntos de interés que sí se van a visitar del total. Si se visitan todos los puntos de interés, entonces $n = k$ y obtenemos que la cantitad total de permutaciones es igual a ${}_nP_r = n!$. Si nuestro conjunto es de 5 ciudades, y se visitarán las 5, entonces el total de permutaciones es 120. Si se tienen 10 ciudades y se visitaran todas, entonces el total de permutaciones aumenta a 3628800. Y si fuesen 15, las permutaciones

incrementarian a un ridículo total de 1307674368000 ¿Con tantas maneras distintas de realizar el recorrido cómo se puede encontrar la mejor? El problema parece simple de resolver, sí, cuando no se consideran tantos puntos de interés. El trabajo presente apunta a encontrar una solución a esta incógnita tomando a los puntos de interés como las capitales de México siendo un total de 32.

Antes de continuar quisiera mencionar que no fui capaz de realizar el cómputo mediante fuerza bruta considerando el total de capitales lo cuál era de esperarse, ya que computar todas las permutaciones requiere memoria y la librería en `julia` disponible para el cómputo de permutaciones depende de la función `factorial` la cual tiene una restricción, no puede ser usada para valores mayores a 20. Por lo que `factorial(21)` ya no computa.

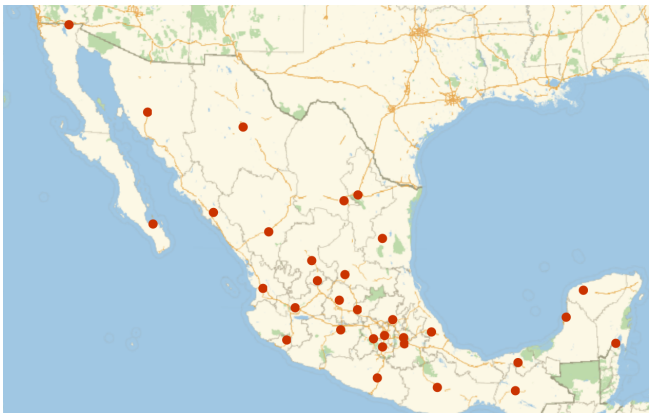

Figura 1: Ubicación de las capitales de México.

Las capitales de México se pueden ver ubicadas en la figura 1 con circulos de color rojo, siendo un total de 32. Las coordenadas fueron obtenidas usando GoogleMaps buscando cada ciudad y tomando sus coordenasas aproximadamente en sus centros. Se implementó una función para guardar la inforación de las capitales, `available_cities` y usarla posteriormente cuando se necesite hacer una selección aleatoria de las mismas.

Comenzaremos tomando 5 ciudades de manera aleatoria, se mostrarán los nombres de las ciudades seleccionadas (en el orden en que se visitarán) junto con sus latitudes y longitudes correspondientes, como se muestra a continuación:

```
# Seleccion aleatoria de ciudades
julia> sample_cities = ts.sample_cities(5);

# Mostrar ciudades seleccionadas
julia> map(x -> x.name, sample_cities)
5-element Vector{String}:
"oaxaca"
"saltillo"
"zacatecas"
"queretaro"
"tlaxcala"

# Mostrar coordenadas de ciudades seleccionadas
julia> map(x -> (x.lat, x.lon), sample_cities)
5-element Vector{Tuple{Float64, Float64}}:
(17.062183511066106, -96.72572385123796)
(25.425170167352245, -101.00211644466016)
(22.772858479171045, -102.57341087527752)
```

```
(20.592088731107133, -100.3918227421049)
(19.314544474512967, -98.23851540921879)
```

El mapeo muestra el nombre de las ciudades en el orden en el que se van a visitar, se comienza en Oaxaca y se termina en Tlaxcala. Aunque este es un viaje redondo, esto significa que después de haber llegado Tlaxcala se debe regresar a Oaxaca nuevamente. En la figura 2 se puede observar el recorrido inicial de esta configuración,

Figura 2: Ruta inicial para cinco capitales comenzando en Oaxaca.

El recorrido mostrado en figura 2 es el realizado si se hiciera caso a la selección aleatoria, pero no se desea eso. Primero se calcularán las permutaciones de la selección aleatoria de capitales guardada en `sample_cities`:

```
# Computando las permutaciones
julia> path_permutations = ts.brute_force(sample_cities, "geo");

# Calculando la distancia de cada ruta
julia> path_permutations_dist = ts.total_distance.(path_permutations, "geo");
```

Teniendo las permutaciones se buscará la ruta con la distancia mínima, y también alguna otra ruta cuya distancia sea igual a la mínima como se muestra a continuación:

```
# Camino de distancia minima
julia> min_path = path_permutations[argmin(path_permutations_dist)]

# Distancia minima
julia> min_dist = minimum(path_permutations_dist)
2250.1837692021245

# Buscando los indices del camino o caminos mas cortos cuya distancia sea igual a la minima encontrada
julia> optimal_indexes = findall(x -> x == min_dist, path_permutations_dist)
10-element Vector{Int64}:
15
19
33
44
59
```

```
62
77
88
102
106

# Encontrando las rutas que coinciden con la distancia minima
julia> map.(x -> x.name, path_permutations[optimal_indexes])
10-element Vector{Vector{String}}:
["oaxaca", "queretaro", "zacatecas", "saltillo", "tlaxcala"]
["oaxaca", "tlaxcala", "saltillo", "zacatecas", "queretaro"]
["saltillo", "zacatecas", "queretaro", "oaxaca", "tlaxcala"]
["saltillo", "tlaxcala", "oaxaca", "queretaro", "zacatecas"]
["zacatecas", "saltillo", "tlaxcala", "oaxaca", "queretaro"]
["zacatecas", "queretaro", "oaxaca", "tlaxcala", "saltillo"]
["queretaro", "oaxaca", "tlaxcala", "saltillo", "zacatecas"]
["queretaro", "zacatecas", "saltillo", "tlaxcala", "oaxaca"]
["tlaxcala", "oaxaca", "queretaro", "zacatecas", "saltillo"]
["tlaxcala", "saltillo", "zacatecas", "queretaro", "oaxaca"]
```

De esta manera nos hemos asegurado de encontrar todas las ocurrencias donde las distancias de los caminos son iguales al mínimo encontrado. Si no se hubiése hecho así y solamente se hubiera aplicado la función `minimum` sí se que habría detectado un mínimo, pero solamente uno. A pesar de que la primera capital en el muestreo aleatorio de capitales fue Oaxaca esto no restringe aque se deba partir de ahí, y también debe recordarse que todas las rutas mostradas con el mismo resultado son rutas de viaje redondo.

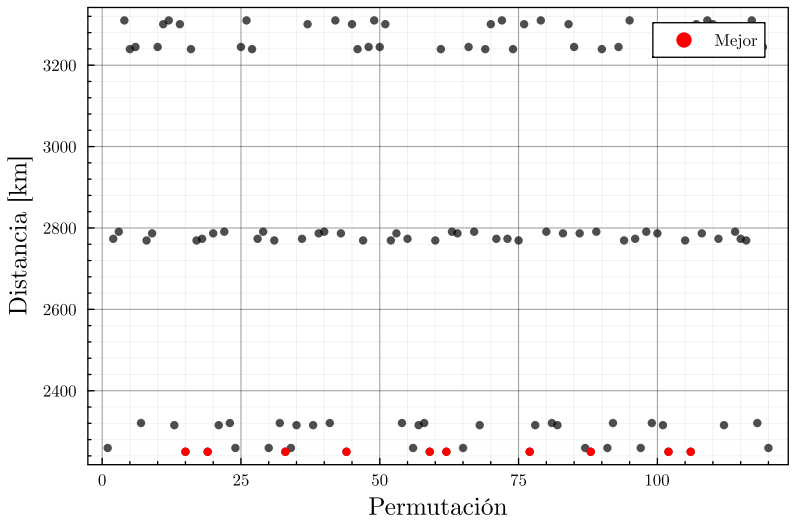

Figura 3: XXX

De la figura 3 se puede observar en marcadores rojos aquellas permutaciones con la misma distancia mínima, mienstras que los de color grisáseo son rutas menos óptimas. En realidad se podría elegir cualquier ruta de las marcadas en rojo pero para motivos de este trabajo vamos a trabajar con la guardada en `min_path`. El recorrido de esta ruta se puede observar en XXX,

Figura 4: XXX

Apéndice

```
module TravellingSalesman

using Random
using Plots
using Combinatorics
using StatsBase


# Generate an initial visiting path order
initial_guess(number_of_coords) = randcycle(number_of_coords)

function initial_guess()
    number_of_cities = 32
    cities, number_of_cities = available_cities()
    visiting_order = randcycle(number_of_cities)
    return cities[visiting_order]
end

function sample_cities(n)
    cities = available_cities()
    return sample(cities, n, replace=false)
end

function available_cities()
    # Coordinates from google maps
    cdmx = CoordCity("cdmx", 19.428262942477954, -99.13307482405206)
    puebla = CoordCity("puebla", 19.040822115386852, -98.20780804508925)
    guadalajara = CoordCity("guadalajara", 20.677308952193865, -103.34688257066186)
    monterrey = CoordCity("monterrey", 25.67766375433872, -100.31367653134096)
    chihuahua = CoordCity("chihuahua", 28.63688593294732, -106.07575156035638)
    merida = CoordCity("merida", 20.96767146389577, -89.62498156651586)
    saltillo = CoordCity("saltillo", 25.425170167352245, -101.00211644466016)
    aguascalientes = CoordCity("aguascalientes", 21.882693391576726, -102.29651596173771)
    hermosillo = CoordCity("hermosillo", 29.081318106282477, -110.95317265344214)
    mexicali = CoordCity("mexicali", 32.62488966123241, -115.45331540638338)
    sanluispotosi = CoordCity("sanluispotosi", 22.152044624927726, -100.97754422094879)
    culiacan = CoordCity("culiacan", 24.808687010506628, -107.39416360809479)
    queretaro = CoordCity("queretaro", 20.592088731107133, -100.3918227421049)
    morelia = CoordCity("morelia", 19.702363913872365, -101.1923888816009)
    durango = CoordCity("durango", 24.024759761413552, -104.670261814078)
    tuxtla = CoordCity("tuxtla", 16.753312818180827, -93.11533318357856)
    xalapa = CoordCity("xalapa", 19.529942056424204, -96.92278227330834)
    tepic = CoordCity("tepic", 21.51212649382402, -104.89139966235507)
    cuernavaca = CoordCity("cuernavaca", 18.92283171215841, -99.2354742591368)
    villahermosa = CoordCity("villahermosa", 17.989794162861205, -92.92869544177128)
    ciudadvictoria = CoordCity("ciudadvictoria", 23.73259400240757, -99.14904253088494)
    pachuca = CoordCity("pachuca", 20.124500636694673, -98.73481509992108)
    oaxaca = CoordCity("oaxaca", 17.062183511066106, -96.72572385123796)
    lapaz = CoordCity("lapaz", 24.161166099496494, -110.3129218770756)
    campeche = CoordCity("campeche", 19.844307251272554, -90.5362438879075)
    chilpancingo = CoordCity("chilpancingo", 17.55248920554459, -99.50078061701078)
    toluca = CoordCity("toluca", 19.29364749241578, -99.65372258157308)
```

```
    chetumal = CoordCity("chetumal", 18.50429901233981, -88.29533434224632)
    colima = CoordCity("colima", 19.24297973037111, -103.72824357123301)
    zacatecas = CoordCity("zacatecas", 22.772858479171045, -102.57341087527752)
    guanajuato = CoordCity("guanajuato", 21.016604105805193, -101.25401186103295)
    tlaxcala = CoordCity("tlaxcala", 19.314544474512967, -98.23851540921879)

    cities = [cdmx, puebla, guadalajara, monterrey, chihuahua, merida, saltillo,
        aguascalientes, hermosillo, mexicali, sanluispotosi, culiacan, queretaro,
        morelia, durango, tuxtla, xalapa, tepic, cuernavaca, villahermosa,
        ciudadvictoria, pachuca, oaxaca, lapaz, campeche, chilpancingo, toluca,
        chetumal, colima, zacatecas, guanajuato, tlaxcala]

    return cities
end

# Cartesian coordinates
mutable struct CoordCartesian
    x
    y
end

# Geo coordinates using latitude and longitude
mutable struct CoordCity
    name    # city name
    lat     # [-π/2, π/2]
    lon     # [-π, π]
end

function distance_function(coordsystem)
    f = Dict("cartesian" => cartesian_distance, "geo" => geo_distance)
    return f[coordsystem]
end

# Distance between two points -> ΔEnergy
cartesian_distance(pointa, pointb) = sqrt((pointb.x - pointa.x)^2 + (pointb.y - pointa.y)^2)

# Distance between two points
function geo_distance(pointa, pointb)
    earth_radius = 6371    # Approx. radius in kilometres
    degtorad = π / 180

    φa, φb = pointa.lat, pointb.lat
    λa, λb = pointa.lon, pointb.lon
    (φa, φb, λa, λb) = (φa, φb, λa, λb) .* degtorad

    # Latitude and longitude differences
    Δφ = (φb - φa)
    Δλ = (λb - λa)

    # Haversine formula
    a = sin(Δφ / 2)^2 + cos(φa) * cos(φb) * sin(Δλ / 2)^2
    c = 2 * atan(sqrt(a), sqrt(1 - a))
    distance = earth_radius * c

```

```
    return distance
end

# Total distance of current path -> Path's Total Energy
function total_distance(coords, coordsystem)
    distfun = distance_function(coordsystem)

    dist = 0
    for (from, to) in zip(coords[begin:end-1], coords[begin+1:end])
        dist += distfun(from, to)
    end

    dist += distfun(coords[begin], coords[end])

    return dist
end

# Simple simulated annealing algorithm
function simulated_annealing(coords, coordsystem; init_temp=30, temp_factor=0.99, max_iter=1000, abstol=1E-3)
    count_coords = length(coords)

    available_random_swaps = 500

    coords_ = deepcopy(coords)
    coords_per_iter = [deepcopy(coords_)]

    cost0 = total_distance(coords_, coordsystem)    # Initial cost
    cost_per_iter = [cost0]

    # Current temperature
    T = init_temp

    for iter = 1:max_iter    # Why 1000?
        T = T * temp_factor

        for _ = 1:available_random_swaps    # Why 500?
            # Choose randomly which coordinates to swap
            r1, r2 = rand(1:count_coords, 2)
            # Swap coordinates
            coords_[r1], coords_[r2] = coords_[r2], coords_[r1]
            # Get new cost
            cost1 = total_distance(coords_, coordsystem)

            if cost1 < cost0
                cost0 = cost1
            else
                # If current cost is not better then gamble! The system can still
                # change with some probability
                probability = rand()
                if probability < exp((cost0 - cost1) / T)
                    cost0 = cost1
                else
                    # If still probability doesnt play into our favor, then undo
                    # the swap
```

```
                    coords_[r1], coords_[r2] = coords_[r2], coords_[r1]
                end
            end
        end

        push!(coords_per_iter, deepcopy(coords_))
        push!(cost_per_iter, cost0)

        # Early stopping condition because a good enough solution was found
        if iter > 1 && abs(cost_per_iter[end] - cost_per_iter[end-1]) <= abstol
            break
        end
    end
    return coords_, coords_per_iter, cost_per_iter
end

function brute_force(coords, coordsystem)
    count_coords = length(coords)
    perms_available = prod(1:count_coords)    # Equivalent to factorial
    perms = []
    for i in 1:perms_available
        perm = nthperm(coords, i)
        push!(perms, perm)
    end
    return perms
end

function brute_force_1(coords, coordsystem)
    count_coords = length(coords)
    indexes_perms = permutations(1:count_coords, count_coords)
    coords_perms = map(x -> coords[x], indexes_perms)
    distances = total_distance.(coords_perms, coordsystem)
    min_distance = argmin(distances)
    return coords_perms[min_distance]
end

function brute_force_3(coords, coordsystem)
    count_coords = length(coords)
    perms_available = prod(1:count_coords)    # Equivalent to factorial
    perms = map(x -> nthperm(coords, x), 1:perms_available)    # Get all permutations
    return perms
end

end # module TravellingSalesman
```

Referencias

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 9 2017.

[2] Simulated annealing. `https://en.wikipedia.org/wiki/Simulated_annealing`. Visitado: 2022-12-21.

[3] Travelling salesman problem. `https://en.wikipedia.org/wikiTravelling_salesman_problem`. Visitado: 2022-12-21.