Universidad Veracruzana

Inteligencia Artificial

Análisis de Algoritmos

Proyecto:
Travelling Salesman Problem

Leonardo Flores Torres

2 de enero de 2023

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 utilizarán.
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 minimice la distancia al recorrer un conjunto de puntos de interés, aunque podría extenderse este concepto a encontrar la ruta que minimice el tiempo al recorrer 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 cantidad total de puntos de interés, y k es la del 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 cantidad 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 ¿Cómo se puede encontrar la mejor ruta con tantas maneras distintas de realizar el recorrido? 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.

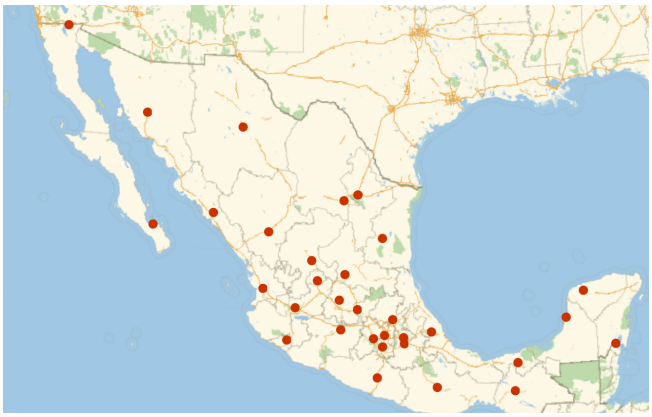

Figura 1: Ubicación de las capitales de México.

Las capitales de México se pueden ver ubicadas en la figura 1 con circulos de color rojo, siendo un total de 32 marcadores. Las coordenadas fueron obtenidas usando GoogleMaps buscando cada ciudad y tomando sus coordenasas aproximadamente en sus centros. Se implementó una función para guardar la información 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 a 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"]    # Permutacion optima
["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 que se 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 a que se deba partir de ahí, y también debe recordarse que todas las rutas mostradas con el mismo resultado son rutas de viaje redondo.

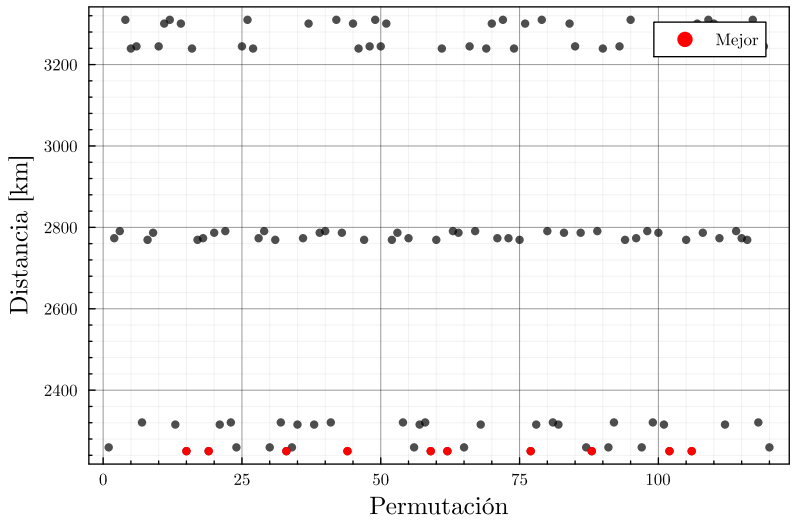

Figura 3: Distancia por permutación considerando 5 capitales.

De la figura 3 se puede observar en marcadores rojos aquellas permutaciones con la misma distancia mínima, mientras que los de color grisáceo 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 figura 4.

Podemos calcular la mejor ruta para esta configuración utilizando el algoritmo de recocido simulado, para esto

Figura 4: Ruta óptima para 5 capitales mediante fuerza bruta.

primero se tiene que dar una suposición inicial que para este caso bien podría coincidir con el orden de las ciudades obtenidas mediantes el muestreo aleatorio. En los casos siguientes se hará una suposición aleatoria inicial del orden de las ciudades para usarlas como entrada en el algoritmo de recocido simulado; aquí se realizará de igual forma.

```
julia> init_guess_indexes = ts.initial_guess(5)
5-element Vector{Int64}:
5
4
1
3
2

julia> init_guess = sample_cities[init_guess_indexes]
5-element Vector{TravellingSalesman.CoordCity}:
TravellingSalesman.CoordCity("tlaxcala", 19.314544474512967, -98.23851540921879)
TravellingSalesman.CoordCity("queretaro", 20.592088731107133, -100.3918227421049)
TravellingSalesman.CoordCity("oaxaca", 17.062183511066106, -96.72572385123796)
TravellingSalesman.CoordCity("zacatecas", 22.772858479171045, -102.57341087527752)
TravellingSalesman.CoordCity("saltillo", 25.425170167352245, -101.00211644466016)
```

El algoritmo tiene un máximo de iteraciones de 1000 pero se puede cambiar si así se requiere o desea aunque se agregó un criterio de terminación temprana. Este criterio detiene el algoritmo si detecta que el valor absoluto de la diferencia de distancias entre el recorrido de dos iteraciones consecutivas es menor a un valor de tolerancia, $|d_i - d_{i-1}| < \varepsilon$. El punto de hacer esto es aprovechar la naturaleza aleatoria de los cambios de ciudades en la ruta, si dos rutas tienen distancias totales similares o la misma, significa que se ha encontrado la ruta óptima deseada y no es necesario seguir computando las iteraciones precedentes que todavía podrían estar pendientes.

```
# Aplicacion de recocido simulado al problema del viajero
julia> best, coords_per_iter, distance_per_iter = ts.simulated_annealing(init_guess, "geo"; init_temp=50,
↪  temp_factor=0.99, max_iter=1000, abstol=1E-5)

# Listado de orden de visita de ciudades por iteracion
julia> map.(x -> x.name, coords_per_iter)
5-element Vector{Vector{String}}:
```

```
["tlaxcala", "queretaro", "oaxaca", "zacatecas", "saltillo"]
["zacatecas", "tlaxcala", "oaxaca", "queretaro", "saltillo"]
["zacatecas", "oaxaca", "tlaxcala", "queretaro", "saltillo"]
["zacatecas", "saltillo", "tlaxcala", "oaxaca", "queretaro"]
["oaxaca", "queretaro", "zacatecas", "saltillo", "tlaxcala"]    # Ruta optima

# Listado del orden de las distancias totales de las rutas encontradas
julia> distance_per_iter
5-element Vector{Float64}:
2769.355722677771
2315.46679146938
2320.7163046789665
2250.1837692021245
2250.1837692021245    # Distancia de la ruta optima
```

Es importante notar que esta ruta mostrada en la línea 77 es la misma ruta que la primera listada en la línea 44, cuando se encontraron todas las permutaciones de las rutas, por lo que la ruta encontrada mediante el algoritmo de recocido simulado será igual a la mostrada en la figura 4. La última ruta mostrada en el listado de las rutas por iteraciones es la ruta óptima que el algoritmo encuentra, y se guarda en la variable `best`. La diferencia de la distancia óptima encontrada por fuerza bruta y la distancia correspondiente a la ruta óptima encontrada por el algoritmo de recocido simulado se muestra en la figura 5.

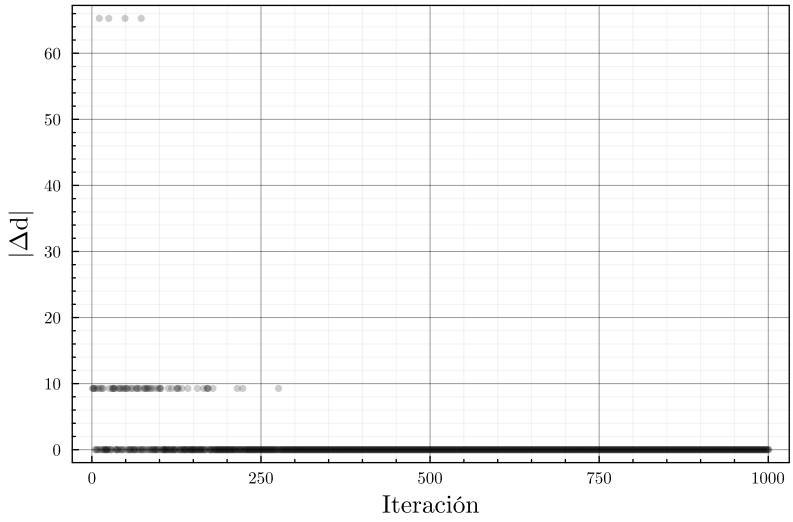

Figura 5: Diferencia entre la distancia óptima obtenida por fuerza bruta y la mejor distancia obtenida mediante el algoritmo.

Se puede observar que hay muchas iteraciones en las que la configuración de la ruta coincide con alguna ruta óptima, y aproximadamente después de la iteración 250 es que se ha convergido a una ruta óptima ya que no se observan cambios en la diferencia de distancias. Por esto es que se decidió agregar una condición para la terminación temprana del algoritmo de recocido simulado.

Se hubiera deseado tomar un número más grande de ciudades para aplicar la comparación entre el algoritmo de recocido simulado y el método por fuerza bruta pero el número de permutaciones aumenta tanto que en el equipo de cómputo utilizado resulta imposible llegar a casos más grandes. anteriormente se había mencionado el problema

con la función `factorial`, pero ahora al tratar de computar las permutaciones para 15 y 20 ciudades, la memoria del ordenador fue insuficiente. Ahora que ya se explicó el proceso para encontrar la ruta óptima se procederá a hacer lo mismo tomando un total de 10 ciudades, el muestreo aleatorio de la lista total de capitales y la suposición inicial para el recorrido se muestran a continuación:

```
julia> sample_cities = ts.sample_cities(10);

julia> map(x -> x.name, sample_cities)
10-element Vector{String}:
"ciudadvictoria"
"saltillo"
"villahermosa"
"durango"
"zacatecas"
"aguascalientes"
"chihuahua"
"puebla"
"hermosillo"
"chetumal"

julia> init_guess_indexes = ts.initial_guess(10);

julia> init_guess = sample_cities[init_guess_indexes];

julia> map(x -> x.name, init_guess)
10-element Vector{String}:
"hermosillo"
"zacatecas"
"ciudadvictoria"
"chetumal"
"durango"
"villahermosa"
"aguascalientes"
"saltillo"
"puebla"
"chihuahua"
```

El recorrido aleatorio inicial para el caso de las 10 ciudades es el mostrado en la figura 6. No se espera que el recorrido inicial sea óptimo de ninguna manera ya que el orden se genera de manera aleatoria. Posterior a generar este recorrido aleatorio se deben de calcular las permutaciones de todas las rutas posibles lo cual se hará a continuación:

```
julia> path_permutations = ts.brute_force(init_guess, "geo");

julia> path_permutations_dist = ts.total_distance.(path_permutations, "geo");

julia> min_path = path_permutations[argmin(path_permutations_dist)];

julia> min_dist = minimum(path_permutations_dist)
5416.112742134113

julia> optimal_indexes = findall(x -> x == min_dist, path_permutations_dist)
11-element Vector{Int64}:
```

Figura 6: Recorrido aleatorio inicial para un total de 10 ciudades.

```
123655
353711
488642
597881
1490556
1516018
1948927
2487212
2633601
3121779
3560158

julia> map.(x -> x.name, path_permutations[optimal_indexes])
11-element Vector{Vector{String}}:
["hermosillo", "durango", "zacatecas", "aguascalientes", "puebla", "villahermosa", "chetumal", "ciudadvictoria", "saltillo", "chihuahua"]    # Permutacion optima
["hermosillo", "chihuahua", "saltillo", "ciudadvictoria", "chetumal", "villahermosa", "puebla", "aguascalientes", "zacatecas", "durango"]
["zacatecas", "durango", "hermosillo", "chihuahua", "saltillo", "ciudadvictoria", "chetumal", "villahermosa", "puebla", "aguascalientes"]
["zacatecas", "aguascalientes", "puebla", "villahermosa", "chetumal", "ciudadvictoria", "saltillo", "chihuahua", "hermosillo", "durango"]
["durango", "hermosillo", "chihuahua", "saltillo", "ciudadvictoria", "chetumal", "villahermosa", "puebla", "aguascalientes", "zacatecas"]
["durango", "zacatecas", "aguascalientes", "puebla", "villahermosa", "chetumal", "ciudadvictoria", "saltillo", "chihuahua", "hermosillo"]
["villahermosa", "chetumal", "ciudadvictoria", "saltillo", "chihuahua", "hermosillo", "durango", "zacatecas", "aguascalientes", "puebla"]
["aguascalientes", "puebla", "villahermosa", "chetumal", "ciudadvictoria", "saltillo", "chihuahua", "hermosillo", "durango", "zacatecas"]
["saltillo", "ciudadvictoria", "chetumal", "villahermosa", "puebla", "aguascalientes", "zacatecas", "durango", "hermosillo", "chihuahua"]
["puebla", "villahermosa", "chetumal", "ciudadvictoria", "saltillo", "chihuahua", "hermosillo", "durango", "zacatecas", "aguascalientes"]
["chihuahua", "saltillo", "ciudadvictoria", "chetumal", "villahermosa", "puebla", "aguascalientes", "zacatecas", "durango", "hermosillo"]
```

Nótese que se tiene una lista de 11 rutas igualmente óptimas, todas con una distancia indicada por `min_dist` de 5416.11 km. En teoría una persona, como el vendedor del problema, podría comenzar por cualquiera de las ciudades marcadas al inicio de esas listas, realizar el recorrido redondo, y habría recorrido la misma distancia al llegar nuevamente a la ciudad de salida. Ahora se debe encontrar la mejor ruta mediante el algoritmo de recocido simulado:

Figura 7: Ruta óptima para 10 capitales mediante fuerza bruta.

```
julia> best, coords_per_iter, distance_per_iter = ts.simulated_annealing(init_guess, "geo"; init_temp=30,
↪  temp_factor=0.99, max_iter=1000);

julia> map(x -> x.name, best)
10-element Vector{String}:
"chihuahua"
"hermosillo"
"durango"
"zacatecas"
"aguascalientes"
"puebla"
"villahermosa"
"chetumal"
"ciudadvictoria"
"saltillo"

julia> distance_per_iter[end]
5416.112742134114
```

Es interesante ver que la ruta óptima encontrada mediante el cómputo de todas las permutaciones y la ruta encontrada mediante el algoritmo de recocido simulado es la misma, con la diferencia en que la primera comienza en Hermosillo y la segunda en Chihuahua pero en realidad el recorrido es el mismo si se considera que debe ser redondo. Como la ruta es la misma, entonces el recorrido visual se verá igual que aquella mostrada en la figura 7.

Para terminar este trabajo se mostrará el recorrido óptimo encontrado para las 32 capitales de México (no se hizo la comparación con el método de fuerza bruta por las razones anteriormente mencionadas):

```
julia> sample_cities = ts.sample_cities(32);
```

```
julia> init_guess_indexes = ts.initial_guess(32);

julia> init_guess = sample_cities[init_guess_indexes];

julia> best, coords_per_iter, distance_per_iter = ts.simulated_annealing(init_guess, "geo"; init_temp=30,
↪  temp_factor=0.99, max_iter=1000);

julia> map(x -> x.name, best)    # Recorrido de la ruta optima
32-element Vector{String}:
"toluca"
"morelia"
"queretaro"
"guanajuato"
"guadalajara"
"aguascalientes"
"zacatecas"
"sanluispotosi"
"ciudadvictoria"
"monterrey"
"saltillo"
"durango"
"chihuahua"
"hermosillo"
"mexicali"
"lapaz"
"culiacan"
"tepic"
"colima"
"chilpancingo"
"oaxaca"
"tuxtla"
"villahermosa"
"chetumal"
"merida"
"campeche"
"xalapa"
"puebla"
"tlaxcala"
"pachuca"
"cdmx"
"cuernavaca"

julia> distance_per_iter[end]    # Distancia de la ruta optima
9161.352094270975
```

La distancia que se obtiene al final del algoritmo permitiendo que se compute el número máximo de iteraciones es 9161.35 km, y la visualización de su respectivo recorrido se puede observar en la figura 8.

Antes de terminar este trabajo se quisiera aclarar un par de puntos importantes respecto a esta implementación:

- Los caminos mostrados solamente reflejan las distancias que existen entre dos capitales no las rutas reales que

Figura 8: Ruta óptima para el recorrido de las 32 capitales de México.

se deberían de tomar. Si una persona pudiera viajar en *línea recta*[1] sobre la superficie de una esféra cruzando los mares de manera indistinta entonces sí que sería la ruta que debería de recorrer.

- La distancia no es una distancia cartesiana como se acostumbra manejar en el mayor de los casos. Se usó una distancia computada mediante la fórmula de Haversine [1, 2, 3] la cual encuentra la distancia entre dos puntos en una esféra.
- El algoritmo utilizado en este trabajo fue adaptado de un código existente [4], se reescribió usando el lenguaje de programación `julia` y se adaptó para cumplir con los requisitos del proyecto.
- Las imágenes no se obtuvieron usando `julia`, para esto se exportaron las coordenadas de las rutas de cada imágen mostrada aquí y se procesaron con el lenguaje de programación `mathematica` el cual tiene funciones para graficar puntos en el mapa a partir de sus coordenadas.

Una idea para complementar este trabajo, la cual es más laboriosa ya que llevaría más tiempo de implementar, sería crear una matriz de adyacencia donde las entradas de dicha matriz fuesen las distancias de rutas reales entre ciudades. De esta manera ya no se necesitaría computar la distancia entre ciudades, estarían guardadas en una matriz y representarían realmente las distancias de carreteras dentro del país. Se tendría que definir una manera de acceder a la matriz a partir de un par de ciudades para buscar la distancia entre ellas, y la matriz sería una cuadrada de 32×32 debido al total de capitales.

[1] Se abusó del significado de esta aseveración. Se hizo solamente con motivos ilustrativos.

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 = 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=nothing)
    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 abstol !== nothing && 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

end # module TravellingSalesman
```

Referencias

[1] Calculate distance, bearing and more between latitude/longitude points. https://en.wikipedia.org/wiki/Haversine_formula. Visitado: 2022-12-24.

[2] Calculate distance, bearing and more between latitude/longitude points. http://www.movable-type.co.uk/scripts/latlong.html. Visitado: 2022-12-25.

[3] Calculate distance between two latitude-longitude points? (haversine formula). https://stackoverflow.com/questions/27928/calculate-distance-between-two-latitude-longitude-points-haversine-formula, 2009. Visitado: 2022-12-25.

[4] ComputationalScientist. Simulated annealing algorithm in python - travelling salesperson problem. https://youtu.be/35fzyblVdmA, 2021. Visitado: 2022-12-26.

[5] 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.

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

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