

**TEMA:** **EL PROBLEMA DEL VIAJANTE DE COMERCIO (TRAVELLING SALESMAN PROBLEM)**

**Materia:** Lenguajes y Autómatas II, PROGRAMACION LOGISTICA FUNCIONAL (PLF)

**Integrantes: Irving Daniel Ventura Hernández**

**ING. SITEMAS COMPUTACIONALES**

**3701**

**°°° EL PROBLEMA DEL VIAJANTE DE COMERCIO (TRAVELLING SALESMAN PROBLEM) °°°**

El primer problema que se va a tratar es bastante conocido dentro de la IA y es conocido como “El problema del viajante de comercio” o por sus siglas en ingles **TSP (Travelling Salesman Problem).** La formulación del problema es sencilla: Sean N ciudades, todas conectas por carretera entre sí y cuyas distancias entre ellas es conocida. Nuestro objetivo es encontrar una ruta que, empezando en una ciudad y acabando en la misma, pase exactamente una vez por cada una de las ciudades (excepto por la primera, que será visitada dos veces) y minimice la distancia recorrida.

Tras este simple enunciado se esconde un problema tremendamente difícil. Tanto que, incluso con los ordenadores más potentes que hoy existen, no se tiene la capacidad de resolverlo. Incluso para un numero de ciudades moderado. Aun así, es posible conseguir aproximaciones muy buenas utilizando diferentes técnicas de IA, algunas de las cuales serán estudiadas más adelante.

Se desea construir un modelo con el cual la computadora pueda procesar. Se inicia con el mapa, si hay varias opciones, Por ejemplo, es posible optar por representar las ciudades mediante un grafo ponderado. En este caso cada nodo del grafo representará una ciudad, y cada arista corresponderá a una carretera con su distancia (peso). Se debe tener en cuenta, que para el caso TSP, la distancia a cubrir desde una ciudad A hasta otra B no tiene por qué ser la misma, que de B hasta A. De todas formas, para simplificar el ejemplo, se supondrá, que las distancias entre 2 ciudades son iguales, independientemente del sentido en el que se vaya.

C

52

D

E

B

A

13

25

45

65

15

15

23

27

32

Otra posible representación es utilizar una tabla (también llamada matriz de adyacencia) como la siguiente:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** |
| **A** | 0 | 25 | 52 | 65 | 23 |
| **B** | 25 | 0 | 13 | 15 | 45 |
| **C** | 52 | 13 | 0 | 15 | 27 |
| **D** | 65 | 15 | 15 | 0 | 32 |
| **E** | 23 | 45 | 27 | 32 | 0 |

Siempre que nos sea posible debemos de elegir la representación más simple e intuitiva a no ser que en términos de eficiencia sea más conveniente utilizar otra estructura de datos más compleja. En el caso que se presenta, parece que la tabla es más interesante, ya que puede ser modelada más fácilmente con un arreglo de dos dimensiones.

La implementación en forma de grafo no aporta nada, ya que se trata de una estructura de datos estática que no cambia durante la ejecución (por ejemplo: un problema en el cual se fueran descubriendo nuevas ciudades, según avance la ejecución), entonces sería interesante usar un grafo como estructura de datos dinámica.

Una vez modelada la información del problema, se necesita una forma de representar la solución, es decir el objetivo. Una forma simple y directa de representarlo es con una lista de ciudades, que cumpla las siguientes premisas.

* La lista comienza y termina por la misma ciudad
* En la lista aparecen todas las ciudades.
* No se repite ninguna ciudad, excepto la primera y la última.

Cualquier lista de ciudades que cumpla las premisas anteriores, es una posible solución. Sin embargo, se sabe que no todas las soluciones son iguales. De hecho, se busca la ruta con menor distancia. Es necesaria una función de evaluación, que informe de la calidad de solución, para poder compararla con otras soluciones. Para TSP se usará la suma de distancia de cada una de las ciudades por las que se va pasando. Por ejemplo, estas son las posibles soluciones y el valor obtenido por la función de evaluación.

* A – B – E – D – C – A (Función de evaluación: 169).
* A – E – D – C – B – A (Función de evaluación: 108).
* A – C – D – E – B – A (Función de evaluación: 169).

1.- Se tiene que generar todas las posibles soluciones.

2.- Comparar cada solución con el resto de las soluciones.

3.- Quedarse con la solución más optima.

Analicemos con detalle el primer paso. ¿Cómo generamos todas las posibles soluciones?, ¿Cuántas soluciones hay?

Una forma de generar las posibles soluciones es partir de una lista de ciudades cualquiera e ir permutándolas entre ellas. Cada permutación será una posible solución al problema. Vamos a suponer que la lista de ciudades es la siguiente:

* A – B – E – D – C – A (Función de evaluación: 169).

Si hacemos el cambio entre las ciudades B y E obtenemos:

* A – E – B – D – C – A (Función de evaluación: 150).

Si efectuáramos el cambio de E y D sobre la primera ruta, obtenemos:

* A – B – D – E – C – A (Función de evaluación: 207).

Y así sucesivamente hasta obtener todas las combinaciones posibles. Pero, ¿cuántas hay? ¡Si tenemos en cuenta N ciudades, tenemos N! combinaciones posibles. N! (Es leído como N factorial) es igual a:

N! n\*(n-1) \*(n-2) \*…\*1

A cada posible solución la llamaremos **estado,** y al conjunto de todas las posibles soluciones lo denominaremos **espacio de estados**. En nuestro conjunto ejemplo con cinco ciudades se obtiene se obtiene 120 posibles soluciones o estados. Un ejemplo manejable por cualquier computadora.

5! = 5 \* 4 \* 3 \* 2 \* 1 = 120

* Con 7 ciudades se tendría 5,040 posibles soluciones.
* Con 10 ciudades se tendría 3,682,800 posibles soluciones.
* Con 20 ciudades se tendría 2,432,902,008,176,640,000 posibles soluciones.

Para hacernos una idea, si cada permutación le ocupara a la computadora un microsegundo con 20 ciudades necesitaríamos 77,147 años para generar todo el espacio de estados. Como se puede ver es que es muy rápido el crecimiento del espacio de estados según el número de ciudades.

**- PROBLEMAS INTRATABLES -**

A este tipo de problemas se les denomina **intratables** y se dice que pertenecen a la clase **NP,** que sin entrar en detalle se dirá que son una clase de problemas cuya complejidad hace que no sean resolubles en un tiempo razonable por computadora.

Por lo tanto, queda claro, que con métodos computacionales clásicos no es posible resolver este problema. Precisamente se trata de cómo abordar este tipo de problemas a través de diferentes técnicas que serán tratadas más adelante. Pero las soluciones serán cuasi-optimas o al menos en un tiempo de solución aceptable.

**°°° EL PROBLEMA DE LA SATISFACTIBILIDAD BOOLEANA °°°**

Tras este largo y estrambótico nombre se encuentra otro problema básico. El problema de la satisfactibilidad booleana o, para abreviar, **SAT (Boolean Satisfiability Problem)** y tiene el honor de ser el primer problema conocido de la clase **NP**. Aun así, fue hace solo 5 décadas, en 1971, cuando se demostró que SAT era un problema intratable.

Antes de enunciar el problema recordemos algunos conceptos básicos de lógica booleana. Una variable booleana es aquella que puede tomar dos valores: verdadero y falso. También se pueden representar como 0 y 1, o como V y F.

Sobre estas variables se puede realizar varias operaciones, pero nos centraremos en las tres principales: conjunción, disyunción y negación.

La conjunción, también conocida como operación AND y representada por el símbolo, es una operación binaria (en el sentido en que operan dos variables) y que devuelve el valor verdadero, solo cuando ambos operadores son verdaderos. La siguiente tabla llamada tabla de verdad, muestra el resultado de operar con la operación conjuntiva (AND).

|  |  |  |
| --- | --- | --- |
| p | q | pɅq |
| V | V | V |
| V | F | F |
| F | V | F |
| F | F | F |

La disyunción u operador se representa con el símbolo V y devuelve el valor verdadero si alguno de los operadores es verdadero, según la siguiente tabla.

|  |  |  |
| --- | --- | --- |
| p | q | pVq |
| V | V | V |
| V | F | V |
| F | V | V |
| F | F | F |

Por último, el operador unario (opera sobre una sola variable) de negación u operador NOT se representa por el símbolo ¬

delante de la variable. Su tabla de verdad se muestra en la siguiente tabla.

|  |  |
| --- | --- |
| p | ¬P |
| V | V |
| V | F |

Podemos construir funciones booleanas cuyo resultado depende de las variables que la componen. Por ejemplo, sea la función:

¬x3

Si representamos el valor verdadero como T y falso como F, las siguientes asignaciones de variables darían como resultado los siguientes valores.

Las funciones booleanas suelen expresarse de forma normalizada para que sea más fácil operar con ellas. Las dos formas habituales son la forma Normal Conjuntiva (FNC) y la forma Normal Disyuntiva (FND). La FNC se expresa como agrupaciones de disyunciones llamadas **cláusulas,** unidaspor conjunciones de la siguiente manera.

En cambio, la FND tiene el siguiente aspecto

Ambas formas son igualmente válidas, pero se usará preferentemente la FNC en mi clase.

Una cláusula puede tener un número cualquiera de variables, y a su vez, una función booleana en FNC o FND, puede tener varias cláusulas como sea necesario.

Finalizando este breve recordatorio de lógica booleana, estando listos para enunciar el problema SAT.

Siendo F una función booleana en FNC con N variables se quiere saber si existe una asignación por las variables, tal que haga verdadera a la función F. Por ejemplo, se considera la siguiente función.

¿Existe una asignación para las variables x1, x2, x3, x4, que haga verdadera la F? En este ejemplo es posible encontrar una asignación de variables sin tener que pensar demasiado.

Por ejemplo, la asignación x1=F, x2=V, x3=V, x4=F, satisface las restricciones del problema.

Ahora si se quiere resolver el problema SAT con una función de tres variables por cláusula (problema conocido como 3-SAT) y 100 variables booleanas, la cosa cambia y se complica un poco. Se plantea un modelo, un objetivo y una función de evaluación.

En este caso el modelo es bastante directo, se puede representar las asignaciones de variables como cadena de valores 0 y 1. Por ejemplo x1=F, x2=V, x3=V, x4=F, estaría representado por la cadena de 0110. Si tiene 100 variables se tendría una cadena de 100 dígitos binarios.

Una cadena cualquiera de N dígitos binarios es candidata a ser una solución y, por lo tanto, se tendrá que ir evaluándolas con una función de evaluación. Para 100 variables, el espacio de estados (número de posibles soluciones) es de 2100, es decir un número astronómicamente alto. Nuevamente se encuentra un problema intratable de clase NP.

Además, encontrar una función de evaluación que permita comparar soluciones, tampoco es fácil. Si se evalúa la función, devolverá un verdadero o falso, pero no se tiene la información de cuantas variables son correctas y cuantas no (si se supiera, entonces significaría que se conoce la solución). Si se compara dos cadenas diferentes que devuelvan el valor *false*, ¿cómo se sabrá cual fue el mejor?

Al ser una función en FNC, se sabe, que la evaluación no puede ser verdadera. Por lo tanto, una función de evaluación, que daría una medida de la calidad de una asignación es aquella que devuelve el número de cláusulas que son verdaderas.

**°°° EL PROBLEMA DE LA PROGRAMACIÓN LINEAL ENTERA °°°**

El problema de la programación lineal entera **(PLE)** consiste en la optimización de una función en la que intervienen variables enteras y, además, hay unas restricciones impuestas por ejemplo sea la función:

Donde x1 y x2 son dos variables enteras, y sean también las siguientes restricciones:

Es decir, se busca una asignación de valores enteros para las dos variables que maximice (o minimice) el valor de la función F y que a su vez se cumplan las restricciones impuestas de ser mayores que cero y que la suma de 7x1 y 4x2 sea menor que 13.

La programación lineal entera es un caso particular de la programación lineal (PL), en la que se exige que todas o algunas variables sean enteras. En el caso de PL las variables son reales. Pese a lo que puede parecer a primera vista, la PLE es un problema más complejo de resolver que la PL.

Para hacerse una idea de la utilidad que se tiene para resolver este tipo de problemas, se plantea un ejemplo mucho más elaborado.

Una empresa fabrica dos tipos de prendas de vestir: camisetas y pantalones. El tiempo necesario para fabricar un pantalón es de 3 horas y se necesitan 4 metros de tela. En el caso de la camiseta se tarda 2 horas y son necesarios 3 metros de tela. También se sabe que la tela está disponible para dentro de una semana y es de 160 metros y con los empleados que hay contratados se pueden invertir 150 horas por semana de trabajo. El dueño de la empresa solicita que a partir del costo de producción y del precio de venta de las prendas, se estime cuantas camisetas y cuantos pantalones tiene que fabricar para obtener el máximo de ganancias.

|  |  |  |
| --- | --- | --- |
|  | Pantalones | Camisetas |
| Horas de producción por prenda | 3 | 2 |
| Metros por prenda | 4 | 3 |
| Costos de producción | 6 | 4 |
| Precio venta | 12 | 8 |

Comencemos buscando un modelo que represente el problema. Para ello se utilizarán dos variables. Se llamarán x1 al número de pantalones y x2 al número de camisetas. Lo que se busca es el máximo beneficio con los recursos de los que se dispone, así que se define en primera instancia el beneficio según la siguiente ecuación.

**Beneficio = (precio de venta – costo de producción) \* unidades vendidas**

Como el beneficio total será la suma de los beneficios de ambas prendas, llegamos a la siguiente ecuación que representa la función de beneficio para este caso en concreto.

Por otro lado, se debe imponer una serie de restricciones. La primera y más obvia es que ambas variables han de ser mayores o iguales a 0. No se puede fabricar un número negativo de prendas.

También se sabe que se dispone de una capacidad de trabajo semanal de 150 horas. Como fabricar un pantalón se invierten 3 horas y 2 en una camiseta, se impone la siguiente restricción al problema.

Lo mismo sucede con la cantidad de metros de tela disponible semanalmente, que es de 160 metros. Como un pantalón necesita 4 metros de tela y una camisa 3 metros, impondrá la siguiente restricción.

En resumen, el modelo para el problema de PLE es:

Restricciones:

El objetivo es encontrar una asignación de valores para las variables x1 y x2 que, cumpliendo con las restricciones impuestas, hagan que la función f tomo el máximo valor posible.

Es posible comprobar fácilmente que la variable x1 está entre 0 y 75. Por lo tanto, el espacio de estados constará de 50 \* 75 = 3,750 posibilidades, no es un número muy elevado. Es perfectamente tratable por un ordenador. Sin embargo, en este problema solo entran en juego dos variables y su rango de valores posibles es muy limitado. Puede demostrarse que para un mayor número de variables y rango de valores el problema es intratable.

La función de evaluación para el problema es muy evidente. Se puede usar la misma función que maximiza y compara las diferentes asignaciones de variables. A mayor valor de la función de evaluación, mejor será la calidad de los valores de las variables.

Un problema relacionado con la PLE es el de programación no lineal **(PNL)** en el que la función o alguna de las restricciones es una función no lineal y las variables toman valores reales (todos los números).

-**OTROS PROBLEMAS –**

Los problemas propuestos hasta ahora no han sido escogidos al azar. Son problemas bien conocidos y estudiados, ya que son problemas canónicos (los problemas canónicos se resuelven siempre con la misma solución, diferentes valores, los canónicos se refiere a que es repetitivo la solución); Es decir cada uno de estos problemas representan y se resuelven de la misma manera que una multitud de otros problemas diferentes. En definitiva, son un modelo que permite atacar la resolución de problemas similares. Por ejemplo, El TSP, es la base de un problema con mucha mayor aplicación práctica: el problema de las rutas de vehículos o Vehicle Routing Problem (VRP). Las distribuciones de mercancías o las empresas de reparto, tratan siempre de optimizar las rutas de reparto, ya que ello trae consigo un gran beneficio económico por el ahorro en recursos; Tiempo, combustible, vehículos y personal. El problema consiste en, dado un numero arbitrario de clientes a los que hay que servir mercancías, calcular las rutas, que minimicen el número de vehículos necesarios y el número de kilómetros necesarios por cada vehículo. Al resolver el problema de las rutas de vehículos, hay que tener presente, que las rutas pueden tener asociadas restricciones como:

* **Carga máxima de vehículos:** dependiendo de la volumétrica y peso de los pedidos de cada cliente, cada vehículo podrá atender a más o menos números de clientes es un viaje.
* **Horario de los clientes:** las rutas tienen que diseñarse de acuerdo a los horarios de cada uno de los clientes, considerando su zona horaria. De nada sirve que la ruta sea optima si él transportista, que suele cobrar por horas, tiene que esperar una hora a que el cliente habrá su comercio, para poder entregar la mercancía.
* **Horario del transportista:** los transportistas pueden tener limitaciones legales em cuanto a la distancia recorrida o tiempo de trabajo que puedan realizar sin descansar.
* **Preferencia:** Algunos clientes pueden ser más importantes que otros y por lo tanto es posible que nos interese servirles antes que a los demás.
* **No De almacenes:** No es lo mismo tener un solo almacén, que varios. Si hay varios almacenes, servirán a cada cliente en concreto siguiendo criterios de cercanía o de existencias de stock en cada almacén.
* **Recarga:** Otro condicionante es si los vehículos cargan mercancía para seguir haciendo entregas. En este caso hay que decidir que los clientes se entreguen en la primera carga y cuales en la segunda y así, sucesivamente.

Evidentemente, si se puede prescindir de un vehiculó, la empresa se ahorra un transportista y a menor número de kilómetros, mayor ahorro de combustible.

A lo largo del semestre se estudiará técnicas que permitan enfrentar problemas con alto grado de complejidad y conseguir soluciones aproximadas y suficientes aceptables.

Pero la IA trata de resolver muchos problemas con alto grado de complejidad y de muy distinto tipo. Estos son algunos de los campos donde se aplican técnicas de la IA con un F1 Score y Accuracy, bastante prometedor.

* **Minería de Datos.**
* **Procesamiento de lenguaje natural**
* **Procesamiento de imágenes**
* **Visión Artificial**
* **Diagnostico medico**
* **Robótica**
* **Entornos industriales**
* **Planificación y control**
* **Video Juegos**
* **Seguridad**
* **Predicciones Financieras**
* **Predicciones Catástrofes**
* **Marketing Especializado**

Encontrar una solución al problema que consistirá en hacer una búsqueda en el árbol de estados o en un grafo para encontrar un nodo que contenga un estado objetivo. En general, el recorrido de un árbol (o grafo) se compone de los siguientes pasos:

1. Seleccionar el nodo raíz y se almacena en una lista que contendrá aquellos nodos que estén pendientes de visitar. A esta lista se llamará **frontera** (también se conoce como **lista abierta**).
2. Se toma un nodo de la lista de frontera y se comprueba si es un nodo objetivo. Si lo es, entonces ha terminado.
3. Se generan todos los hijos del nodo seleccionado en el paso dos, aplicando los operadores que se han definido. A esto se le conoce como **expandir nodo**, se añaden los nuevos nodos hijo a la lista de nodos frontera.
4. Volver al paso dos, hasta que la lista de nodos frontera esté vacía.

Si en lugar de un árbol se está trabajando con un grafo, el proceso cambia ligeramente.

1. Seleccionar el nodo raíz y se almacena en la lista de nodos frontera.
2. Se toma un nodo de la lista de nodos frontera y comprueba si es un nodo objetivo. Si lo es, se ha terminado. Además, se almacena el nodo en una lista llamada **visitados**, que contiene todos los nodos visitados hasta el momento.
3. Se generan todos los nodos hijos del nodo seleccionado en el paso dos, aplicando los operadores que se han definido. Para cada hijo se comprueba que no esté en la lista de nodos visitados, y si no está, se añade a la lista de nodos frontera.
4. Volver al paso dos, hasta que la lista de nodos frontera esté vacía.

**°°° BÚSQUEDA EN AMPLITUD °°°**

El proceso de búsqueda que se acaba de presentar es un procedimiento genérico, que puede dar lugar a variaciones. La primera que se va a presentar se llama **búsqueda en amplitud** o **Breadth First Search** (BFS). Esta búsqueda recorre el árbol por niveles. Es decir:

* Primero se visita el nodo raíz.
* Seguidamente se visitan todos sus hijos.
* Para cada hijo en el paso anterior, se visitan todos sus hijos, y así sucesivamente.

**Nota:** Se queda con la primera solución que encuentra.

En este recorrido se implementa en la práctica usando una cola FIFO (First Int First Out) para la lista de nodos frontera. Una cola FIFO es una estructura de datos en la que se puede hacer dos operaciones principales: Almacenar un dato y usar un dato. La oportunidad de las colas FIFO es cuando se saca un dato, este se extrae en el mismo orden en el que se ha almacenado. A continuación, se muestra el algoritmo de búsqueda en amplitud en pseudocódigo.

**Pseudocódigo búsqueda en amplitud**

**nodo\_inicial = estado\_inicial**

**nodos\_frontera = Cola FIFO**

**nodos\_visitados = Lista**

**almacenar nodos\_frontera no vacíos:**

**nodo\_actual = extraer un nodo de nodos\_frontera**

**si nodo\_actual == solución**

**salir con la solución**

**introducir nodo\_actual en nodos visitados**

**por cada operador:**

**nodo\_hijo no esté en nodos\_visitados ni en nodos\_frontera:**

**introducir nodo\_hijo en nodo\_frontera**

Se continua con la práctica del algoritmo haciendo un programa para resolver el problema del puzzle lineal. Es una sencilla implementación de la estructura de datos del árbol del que se debe hablar más adelante.

*CODIGO ARBOL*

class Nodo:

    def \_\_init\_\_(self, datos, hijos = None):

        self.datos = datos

        self.hijos = None

        self.padre = None

        # Coste del peso acumulado del arbol

        self.coste = None

        self.set\_hijos(hijos)

    def set\_hijos(self, hijos):

        self.hijos = hijos

        if self.hijos != None:

            for h in self.hijos:

                h.padre = self

    def get\_hijos(self):

        return self.hijos

    def get\_padre(self):

        return self.padre

    def set\_padre(self, padre):

        self.padre = padre

    def set\_datos(self, datos):

        self.datos = datos

    def get\_datos(self):

        return self.datos

    def set\_costo(self, costo):

        self.costo = costo

    def get\_costo(self):

        return self.costo

    def igual(self, nodo):

        if self.get\_datos() == nodo.get\_datos():

            return True

        else:

            return False

    def en\_lista(self, lista\_nodos):

        en\_la\_lista = False

        for n in lista\_nodos:

            if self.igual(n):

                en\_la\_lista = True

        return en\_la\_lista

    def \_\_str\_\_(self):

        return str(self.get\_datos())

El uso de la clase nodo no es sencillo. El constructor acepta como primer parámetro un dato que será almacenado en el nodo (en principio de cualquier tipo). Opcionalmente acepta un segundo parámetro con una lista de hijos.

También es posible usar los siguientes métodos:

**set\_hijos-**Asigna al nodo la lista de hijos que son pasados por parámetro.

**get\_hijos-**Retorna una lista con los hijos del nodo.

**get\_padre-**Retorna el nodo padre.

**set\_padre(padre)-**Asigna el nodo padre a este nodo.

**set\_datos(dato)-**Asigna un dato al nodo.

**get\_datos()-**Devuelve el dato almacenado en el nodo.

**set\_peso()-**Asigna un peso al nodo dentro del árbol.

**get\_peso()-**Devuelve el peso del nodo dentro del árbol.

**En\_lista(lista\_nodos)-** Devuelve true si el dato contenido en el nodo es igual a alguno de los nodos contenidos en la lista de nodos pasada como parámetro. El código que resuelve el problema del puzzle lineal es el siguiente:

*CODIGO PUZZLE LINEAL (BFS)*

# PuzleLineal con Busqueda en Amplitud

from arbol import Nodo

def buscar\_solucion\_BFS(estado\_inicial, solucion):

    solucionado = False

    nodos\_visitados = []

    nodos\_frontera = []

    nodoInicial = Nodo(estado\_inicial)

    # Agregar elementos al arreglo "append"

    nodos\_frontera.append(nodoInicial)

    while (not solucionado) and len(nodos\_frontera) != 0:

        # Son los que estan hasta arriba "pop"

        nodo = nodos\_frontera.pop(0)

        # Extraer el nodo y añadirlo a visitados

        nodos\_visitados.append(nodo)

        if nodo.get\_datos() == solucion:

            # Si la solucion encontrada va ser

            solucionado = True

            return nodo

        else:

            # Exapndir nodos\_hijo

            dato\_nodo = nodo.get\_datos()

            # Operador Izquierdo

            hijo = [dato\_nodo[1], dato\_nodo[0], dato\_nodo[2], dato\_nodo[3]]

            hijo\_izquierdo = Nodo(hijo)

            # Se hace comparacion

            if not hijo\_izquierdo.en\_lista(nodos\_visitados) and not hijo\_izquierdo.en\_lista(nodos\_frontera):

                nodos\_frontera.append(hijo\_izquierdo)

            # Operador central

            hijo = [dato\_nodo[0], dato\_nodo[2], dato\_nodo[1], dato\_nodo[3]]

            hijo\_central = Nodo(hijo)

            if not hijo\_central.en\_lista(nodos\_visitados) and not hijo\_central.en\_lista(nodos\_frontera):

                nodos\_frontera.append(hijo\_central)

            # Operador derecho

            hijo = [dato\_nodo[0], dato\_nodo[1], dato\_nodo[3], dato\_nodo[2]]

            hijo\_derecho = Nodo(hijo)

            if not hijo\_derecho.en\_lista(nodos\_visitados) and not hijo\_derecho.en\_lista(nodos\_frontera):

                nodos\_frontera.append(hijo\_derecho)

            # Es dentro de un array

            nodo.set\_hijos([hijo\_izquierdo, hijo\_central, hijo\_derecho])

if \_\_name\_\_ == "\_\_main\_\_":

    estado\_inicial = [4, 2, 3, 1]

    solucion = [1, 2, 3, 4]

    nodo\_solucion = buscar\_solucion\_BFS(estado\_inicial, solucion)

    # Mostrar el resultado

    resultado = []

    nodo = nodo\_solucion

    while nodo.padre != None:

        # Es una función "get\_datos()"

        resultado.append(nodo.get\_datos())

        nodo = nodo.padre

    resultado.append(estado\_inicial)

    resultado.reverse()

    print(resultado)

La ejecución de programa nos mostrara el siguiente resultado:

Listo, ya quedo, eres todo un pro

[[4, 2, 3, 1], [2, 4, 3, 1], [2, 3, 4, 1], [2, 3, 1, 4], [2, 1, 3, 4], [1, 2, 3, 4]]

Que es la secuencia de movimientos que realiza el programa para resolver el puzle.

La función encargada de implementar la búsqueda en amplitud se llama **buscar\_solucion\_BFS ()** que es invocada con dos parámetros. El primero es el estado inicial, que en el ejemplo es: [4,2,3,1]. El segundo parámetro es el estado objetivo, es decir, la solución: [1,2,3,4]. La función retorna con el nodo solución si lo ha encontrado. Para conocer el camino concreto desde el nodo raíz al nodo objetivo, solo hay que recorrer el camino en sentido inverso, desde el nodo objetivo. Como0 cada nodo tiene asignado un padre, solo se tiene que usar la función **get\_padre ()** para recorrer el camino.

Dentro de la función **buscar\_solucion\_BFS (),** lo primero que hace es inicializar las listas de nodos frontera y nodos visitados. También se crea el nodo inicial con el valor del primer parámetro de la función y se añade a la lista de nodos frontera.

La siguiente porción de código es un bucle y se repite mientras haya elementos en la lista de nodos frontera o no se haya alcanzado una solución. El primer paso es extraer el primer nodo pendiente de la cola de nodos frontera e insertarlo en la lista de elementos visitados. Si el nodo es una solución, se sale del bucle devolviendo el nodo. Si no, se expande el nodo aplicando los operadores **I, C, D** y se generan sus hijos, si los hijos no están en la lista de nodos visitados, se añaden a la cola de nodos frontera.

La búsqueda en amplitud tiene algunas propiedades que vale la pena señalar. Cabe preguntarse si la búsqueda en amplitud encontrará siempre una solución. En efecto, es fácil comprobar que, si existe una solución, la búsqueda en amplitud acabará encontrándola (otra cuestión diferente en cuanto a tiempo). Cuando un algoritmo se sabe que encontrará una solución, si se deja el tiempo necesario, se dice que el algoritmo es **completo** y se refiere a esta característica como **completitud** de un algoritmo. Como se ve todos los algoritmos que tratarán son completos.

Sabiendo que siempre se encontrará una solución, otra cuestión que se puede plantear, es, si la solución que se encuentra será la óptima. Si lo es, se dice que el algoritmo es **óptimo** y se refiere a esta característica como **optimalidad** que es una mala traducción de la palabra inglesa **optimality**. Pero, ¿es óptima la búsqueda en amplitud?

Antes de continuar es necesario analizar el significado de la palabra óptimo, que depende del problema que se está tratando y del objetivo que se está buscando. Si se toma como ejemplo el viaje de avión. El objetivo es conseguir el mínimo número de trasbordos posibles. Por lo tanto, la solución es óptima si no es posible encontrar una solución diferente con un número de trasbordos mejor.

Sin embargo, el ejemplo del viaje por carretera, el objetivo era otro: reducir el número de kilómetros. Por lo tanto, una solución será óptima, si no es posible encontrar una ruta con el menor número de kilómetros.

En el caso del puzzle lineal, se podría plantear que una solución posible optima, será aquella que sea diferente y lo resuelva en un número menor de movimientos. Si se observa el árbol, por cada movimiento, se baja un nivel en el árbol. Así, a cuanta menos profundidad se encuentre la solución, menos movimientos habrán sido necesarios para llegar a ella. Como la búsqueda en amplitud recorre el árbol por niveles, queda claro que, en este caso, la solución que se encuentra será óptima.

¿Y con el problema del viajante por carretera?, ¿ocurre igual? Imaginemos una red de carreteras ficticias como la siguiente.

Ciudad 1

Ciudad 2

Ciudad 3

Ciudad 4

Ciudad 5

Ciudad 6

346

110

97

201

210

355

603

617

309

El objetivo es llegar desde la ciudad uno a la ciudad cinco, cubriendo la menor distancia posible. De nuevo usando la búsqueda en amplitud, cada paso por una ciudad, supone bajar un nivel en el árbol, por lo que parece que tiene sentido que, a menor número de ciudades recorridas, mejor solución. Sin embargo, analicemos estas dos soluciones:

* Ciudad uno > ciudad cinco (827 kilómetros).
* Ciudad uno > ciudad cuatro > ciudad cinco (521 kilómetros).

Usando una búsqueda en amplitud se hubiera llegado a la primera solución que atraviesa dos ciudades con 827 kilómetros, (dos niveles del árbol). Sin embargo, si se hubiera permitido que el algoritmo descendiera un nivel más, hubiera encontrado una solución mejor con 521 kilómetros, aunque atraviese una ciudad más. Por lo tanto, en el caso de la ruta por carretera, el algoritmo de búsqueda en amplitud no es óptimo.

En general, es posible decidir, que es óptimo si el costo de la función de la avaluación es un valor decreciente directamente relacionado con la profundidad dentro del recorrido del árbol.

A la hora de decidir por un algoritmo u otro, es conveniente conocer su comportamiento en cuanto al tiempo de ejecución y cantidad de memoria que necesita. El tiempo de ejecución necesario depende directamente del número de nodos generados. A esto se le llama **complejidad temporal**. Si el árbol tiene un factor de ramificación b, del nodo raíz, saldrán hijos b. En el siguiente nivel se tendrá b2, y así sucesivamente. En general, para profundidad **d** se tendrá:

Número de nodos = b+b2+b3+…+bd

Generalmente, se utiliza una notación especial para expresar la complejidad de un algoritmo: la cota superior asintótica, es mejor conocida como notación O, (notación O grandota). Sin entrar en detalle se dirá que se corresponde a una función que se analiza cuando el argumento tiende al infinito. Para este caso se dirá que el algoritmo de búsqueda en amplitud tiene una complejidad temporal O(bd).

En cuanto al uso de memoria, que se llamará **completitud espacial**, la búsqueda en amplitud mantiene en memoria cada uno de los nodos expandidos. Además, al usar dos estructuras de datos se tendrá una complejidad O(BD) para almacenar los nodos frontera y O(BD-1) para los nodos visitados.

La notación O es muy útil para comparar la complejidad de dos algoritmos diferentes.

Antes de seguir avanzando en cosas profesionales, se resolverá el problema de los vuelos en el ejercicio anterior, usando la búsqueda en amplitud.

*CODIGO VUELOS (BFS)*

# Vuelos con Busqueda en Amplitud

from arbol import Nodo

def buscar\_solucion\_bfs(conexiones, estado\_inicial, solucion):

    solucionado = False

    nodos\_visitados = []

    nodos\_frontera = []

    nodo\_inicial = Nodo(estado\_inicial)

    nodos\_frontera.append(nodo\_inicial) # Si no esta, lo insertamos en nodos frontera

    while(not solucionado and len(nodos\_frontera) != 0):

        nodo = nodos\_frontera[0] # Se empuja el resto de los nodos

        # Extraer nodo y añadirlo a visitados

        nodos\_visitados.append(nodos\_frontera.pop(0))

        if nodo.get\_datos() == solucion:

            # Solucion encontrada

            solucionado = True

            return nodo

        else:

            # Expandir nodos hijo (ciudades con conexion)

            dato\_nodo = nodo.get\_datos()

            lista\_hijos = []

            for un\_hijo in conexiones[dato\_nodo]:

                hijo = Nodo(un\_hijo)

                lista\_hijos.append(hijo)

                if not hijo.en\_lista(nodos\_visitados) and not hijo.en\_lista(nodos\_frontera):

                    nodos\_frontera.append(hijo)

            nodo.set\_hijos(lista\_hijos)

if \_\_name\_\_ == '\_\_main\_\_':

    conexiones = {

        'EDO.MEX':{'QRO', 'SLP', 'SONORA'},

        'PUEBLA':{'HIDALGO', 'SLP'},

        'CDMX':{'MICHOACAN'},

        'MICHOACAN':{'SONORA'},

        'SLP':{'QRO', 'PUEBLA', 'EDO.MEX', 'SONORA', 'GUADALAJARA'},

        'QRO':{'EDO.MEX', 'SLP'},

        'HIDALGO':{'PUEBLA', 'GUADALAJARA', 'SONORA'},

        'GUADALAJARA':{'HIDALGO', 'SLP'},

        'MONTERREY':{'SONORA'},

        'SONORA':{'MONTERREY', 'HIDALGO', 'SLP', 'EDO.MEX', 'MICHOACAN'}

    }

estado\_inicial = 'EDO.MEX'

solucion = 'HIDALGO'

nodo\_solucion = buscar\_solucion\_bfs(conexiones, estado\_inicial, solucion)

# Mostrar Resultado

resultado = []

nodo = nodo\_solucion

# Recorrer Arbol

while nodo.get\_padre() != None:

    resultado.append(nodo.get\_datos())

    nodo = nodo.get\_padre()

resultado.append(estado\_inicial)

resultado.reverse()

print(resultado)

        # Resultado: Estado de mexico, Sonora e Hidalgo

Al algoritmo se le ha pedido que encuentre la mejor secuencia de transbordos para ir del Estado de México, hacia Hidalgo y arrojó la siguiente propuesta:

['EDO.MEX', 'SONORA', 'HIDALGO']

Se observa que la solución óptima (según el objetivo que se ha definido), ya que, aunque con 3 conexiones hay otras soluciones posibles, no se encontrará ninguna con menos de tres.

Para representar las conexiones, se usa la estructura de datos llamada Diccionario de Python, que permite de manera sencilla representar todas las conexiones. A la hora de generar los nodos hijo, solo se tiene que obtener los elementos almacenados en el diccionario que corresponde con la ciudad que se está examinando.

**°°° BÚSQUEDA EN PROFUNDIDAD °°°**

La **búsqueda en profundidad** o **Depth First Search (DFS)** recorre el árbol/grafo de forma diferente a la búsqueda en amplitud. En lugar de ir visitando todos los nodos de un mismo nivel, va descendiendo hasta la profundidad máxima de la rama y cuando llega al nodo más profundo continua de la siguiente rama.

**Nota:** Va buscando las posibles soluciones para comparar cual es mejor. Pero es muy costosa.

Desde el punto de vista de la implementación, la búsqueda en profundidad se diferencia en amplitud en la que se utiliza una pila **LIFO (Last In First Out).**

En vez de una cola FIFO. Una pila LIFO tiene dos operaciones: una para introducir un dato y otra para extraer un dato de la pila. A diferencia de la cola, el dato que se extrae de la pila es el último que se introdujo. Se puede imaginar como una pila de datos como una pila de platos. Cada vez que se amontona uno, quedará en la parte superior del montón. Para tomar alguno, siempre se tendrá que tomar el que se encuentre en la parte superior, es decir el ultimo que se agregó.

CODIGO DE PUZLELINEAL (DFS)

# Puzle Lineal con busqueda en profundidad

from arbol import Nodo

def buscar\_solucion\_DFS(estado\_incial, solucion):

    solucionado = False

    nodos\_visitados = []

    nodos\_frontera = []

    nodoInicial = Nodo(estado\_incial)

    nodos\_frontera.append(nodoInicial)

    while (not solucionado) and len(nodos\_frontera) !=0:

        nodo = nodos\_frontera.pop()

        # Extraer nodo y añadirlo

        nodos\_visitados.append(nodo)

        if nodo.get\_datos() == solucion:

            solucionado = True

            return nodo

        else:

            #expandir los nodos hijo

            dato\_nodo = nodo.get\_datos()

            # Operador Izquierdo

            hijo = [dato\_nodo[1], dato\_nodo[0], dato\_nodo[2], dato\_nodo[3]]

            hijo\_izquierdo = Nodo(hijo)

            if not hijo\_izquierdo.en\_lista(nodos\_visitados) \

                and not hijo\_izquierdo.en\_lista(nodos\_frontera):

                nodos\_frontera.append(hijo\_izquierdo)

            # Operador Central

            hijo = [dato\_nodo[0], dato\_nodo[2], dato\_nodo[1], dato\_nodo[3]]

            hijo\_central = Nodo(hijo)

            if not hijo\_central.en\_lista(nodos\_visitados) \

                and not hijo\_central.en\_lista(nodos\_frontera):

                nodos\_frontera.append(hijo\_central)

            # Operador Derecho

            hijo = [dato\_nodo[0], dato\_nodo[1], dato\_nodo[3], dato\_nodo[2]]

            hijo\_derecho = Nodo(hijo)

            if not hijo\_derecho.en\_lista(nodos\_visitados) \

                and not hijo\_derecho.en\_lista(nodos\_frontera):

                nodos\_frontera.append(hijo\_derecho)

            nodo.set\_hijos([hijo\_izquierdo, hijo\_central, hijo\_derecho])

if \_\_name\_\_ == "\_\_main\_\_":

    estado\_incial = [4,2,3,1]

    solucion = [1,2,3,4]

    nodo\_solucion = buscar\_solucion\_DFS(estado\_incial, solucion)

    # Mostrar Resultado

    resultado = []

    nodo = nodo\_solucion

    while nodo.get\_padre() != None:

        resultado.append(nodo.get\_datos())

        nodo = nodo.get\_padre()

    resultado.append(estado\_incial)

    # Recorrer el arbol de forma inversa

    # Para poder sacarlo

    resultado.reverse()

    print(resultado)

Resultado:

[[4, 2, 3, 1], [4, 2, 1, 3], [4, 1, 2, 3], [4, 1, 3, 2], [4, 3, 1, 2], [3, 4, 1, 2], [3, 4, 2, 1], [3, 2, 4, 1], [3, 2, 1, 4], [3, 1, 2, 4], [1, 3, 2, 4], [1, 2, 3, 4]]

Que es un resultado visiblemente peor que el obtenido con la búsqueda de amplitud. Esto indica que la entrada de esta búsqueda no es óptima, como en el caso de la búsqueda en amplitud. Se dice que es **subóptima.** Otro problema que presenta el algoritmo, es que, si es árbol de búsqueda no está acotado en profundidad, es decir, es definitivamente profundo, nunca saldrá de la primera rama. En cuanto a la complejidad temporal, un árbol con profundidad **p** tiene una complejidad **O ().** Recordemos que la complejidad de la búsqueda en amplitud es **O (),** donde **d** es la profundidad en la que se encuentra la solución. Como en la mayoría de los casos **p > d,** que se pueda afirmar que la complejidad de la búsqueda en profundidad será, para la mayoría de los casos, mayor que la búsqueda en amplitud (o en el mejor de los casos igual).

¿Por qué presentar entonces la búsqueda en profundidad, si parece no tener ninguna ventaja sobre la búsqueda en amplitud? Deliberadamente no se ha hablado aun de la complejidad espacial.

En los seudocódigos de ambas búsquedas se ha usado una lista para almacenar los nodos visitados. Se ha hecho así para no perder generalidad y que el algoritmo fuera usable tanto en arboles como en grafos. Ahora bien, si el espacio de búsqueda esta descrito por un árbol y no por un grafo, se puede prescindir de una lista de nodos visitados. En la búsqueda en profundidad, si el espacio de búsqueda está representado por un árbol. La complejidad espacial se reduce a **O(bm),** lo que supone un costo de almacenamiento muy inferior al usado en la búsqueda en amplitud. Esto es debido a que cuando se explora una rama, sus nodos quedan eliminados de la lista frontera. Mientras que en la búsqueda en amplitud es necesarios mantener almacenados los nodos frontera de todas las ramas a la vez.

Para hacernos una idea se tomará como ejemplo el puzle lineal. Si se supone que un nodo necesita 1 kilobyte de almacenamiento (incluyendo las estructuras propias de Python) la exploración de un árbol (es decir, sin lista de nodos visitados), hasta el nivel 10 se necesita unos 60 MB de memoria RAM para la búsqueda en amplitud, mientras que la misma búsqueda usando el algoritmo en profundidad necesita unos 90KB. La diferencia es más que apreciable.

Si la búsqueda se realiza usando la lista de visitados, la búsqueda es también completa. En caso de prescindir de almacenar los nodos visitados en aras de preservar el uso de memoria, el algoritmo no es completo, ya que podría entrar en bucles durante la búsqueda.

Por las características propias de la búsqueda en profundidad, es posible una implementación alternativa usando recursividad. El siguiente seudocódigo muestra la implementación de forma general.

***Funcion\_DFS\_REC (nodo, solución, visitados):***

***Añadir nodo visitados***

***Si nodo == solución***

***Salir con solución***

***Si no:***

***Por cada operador:***

***Nodo\_hijo = operador(nodo)***

***Si nodo\_hijo no está en visitados:***

***S = DFS\_rec (nodo\_hijo, solución, visitados)***

***Salir con s***

***Nodo = estado inicial***

***Visitados = Lista***

***Solución = solución***

***S = DFS\_rec (nodo, solución, visitados)***

El esquema expresado en el seudocódigo se puede llevar a la implementación correcta para resolver el puzle lineal.

CODIGO DE PUZLELINEAL (DFS.REC)

# Puzle lineal con búsqueda en profundidad recursiva

from arbol import Nodo

def buscar\_solucion\_DFS\_Rec(nodo\_inicial, solucion, visitados):

    visitados.append(nodo\_inicial.get\_datos())

    if nodo\_inicial.get\_datos() == solucion:

        return nodo\_inicial

    else:

        # Expandir nodos sucesores (hijos)

        dato\_nodo = nodo\_inicial.get\_datos()

        hijo = [dato\_nodo[1], dato\_nodo[0], dato\_nodo[2], dato\_nodo[3]]

        hijo\_izquierdo = Nodo(hijo)

        hijo = [dato\_nodo[0], dato\_nodo[2], dato\_nodo[1], dato\_nodo[3]]

        hijo\_central = Nodo(hijo)

        hijo = [dato\_nodo[0], dato\_nodo[1], dato\_nodo[3], dato\_nodo[2]]

        hijo\_derecho = Nodo(hijo)

        nodo\_inicial.set\_hijos([hijo\_izquierdo, hijo\_central, hijo\_derecho])

        for nodo\_hijo in nodo\_inicial.get\_hijos():

            if not nodo\_hijo.get\_datos() in visitados:

                # Llamada Recursiva

                sol = buscar\_solucion\_DFS\_Rec(nodo\_hijo, solucion, visitados)

                if sol != None:

                    return sol

        return None

if \_\_name\_\_ == "\_\_main\_\_":

    estado\_inicial = [4, 2, 3, 1]

    solucion = [1, 2, 3, 4]

    nodo\_solucion = None

    visitados = []

    nodo\_inicial = Nodo(estado\_inicial)

    nodo = buscar\_solucion\_DFS\_Rec(nodo\_inicial, solucion, visitados)

    # Mostrar Resultado

    resultado = []

    while nodo.get\_padre() != None:

        resultado.append(nodo.get\_datos())

        nodo = nodo.get\_padre()

    resultado.append(estado\_inicial)

    resultado.reverse()

    print(resultado)

Visto que la búsqueda en profundidad es ventajosa en cuanto al uso de uso de memoria, se puede intentar mejorar el resto de los aspectos para intentar quedarse con lo mejor de la búsqueda en profundidad y la búsqueda en amplitud.

Uno de los problemas de búsqueda en profundidad se da cuando el espacio de estados es finito. En este escenario no es posible que el algoritmo de con buena solución (a no ser que tenga la tremenda suerte de que se encuentre en la primera rama del árbol).

Además, el algoritmo no terminara nunca. Para evitar esto, se puede hacer una modificación al algoritmo para evitar que descienda más de un numero prefijado de niveles. Así se evita que quede en ejecución indefinidamente. Se trata de la **búsqueda en profundidad limitada.**

***Función DFS\_limitado(nodo, solución, visitados, limite):***

***Añadir nodo a visitados***

***Si nodo == solución:***

***Salir con solución***

***Si no si limite == 0:***

***Salir con corte\_limite***

***Si no:***

***Por cada operador:***

***Nodo\_hijo = operador(nodo)***

***Si no\_hijo no está en visitados:***

***S = DFS rec(nodo\_hijo, solución, visitados, limite-1)***

***Salir con s***

***Nodo = estado\_inicial***

***Visitados = lista***

***Solución = solución***

***Limite = limite\_niveles***

***S = DFS\_limitado (nodo, solución, visitados, limite)***

# Puzle Lineal con busqueda en profundidad limitada

from arbol import Nodo

def buscar\_solucion\_BFS\_rec(nodo\_inicial, solucion, visitados):

    visitados.append(nodo\_inicial.get\_datos())

    if nodo\_inicial.get\_datos() == solucion:

        return nodo\_inicial

    else:

        #Expandir los nodos sucesores (hijos)

        dato\_nodo = nodo\_inicial.get\_datos()

        hijo = [dato\_nodo[1], dato\_nodo[0], dato\_nodo[2], dato\_nodo[3]]

        hijo\_izquierdo = Nodo(hijo)

        hijo = [dato\_nodo[0], dato\_nodo[2], dato\_nodo[1], dato\_nodo[3]]

        hijo\_central = Nodo(hijo)

        hijo = [dato\_nodo[0], dato\_nodo[1], dato\_nodo[3], dato\_nodo[2]]

        hijo\_dercho = Nodo(hijo)

        nodo\_inicial.set\_hijos([hijo\_izquierdo, hijo\_central, hijo\_dercho])

        for nodo\_hijo in nodo\_inicial.get\_hijos():

            if not nodo\_hijo.get\_datos() in visitados:

                #Llamada recursiva

                sol = buscar\_solucion\_BFS\_rec(nodo\_hijo, solucion, visitados)

                if sol != None:

                    return sol

        return None

if \_\_name\_\_ == "\_\_main\_\_":

    estado\_inicial = [4, 2, 3, 1]

    solucion = [1, 2, 3, 4]

    nodo\_solucion = None

    visitados = []

    nodo\_inicial = Nodo(estado\_inicial)

    nodo = buscar\_solucion\_BFS\_rec(nodo\_inicial, solucion, visitados)

    # Mostrar Resultado

    resultado = []

    # Estructura de control

    while nodo.get\_padre() != None:

        resultado.append(nodo.get\_datos())

        nodo = nodo.get\_padre()

    # Imprimir resultado

    resultado.append(estado\_inicial)

    resultado.reverse()

    print(resultado)

            #Factorial de 4! = 24

Partiendo de esta mejora se puede conseguir, además, que el algoritmo sea optimo. La técnica se llama **búsqueda con profundidad iterativa.** La idea es ir repitiendo la búsqueda de forma iterativa, pero incrementando el nivel de profundidad en cada iteración.

***Función DFS\_prof\_iter(nodo, solución):***

***Para límite de 1 a ∞***

***Visitados = lista***

***S = DFS\_limitado(nodo, solución, limite)***

***Si s ≠* corte\_limite:**

**Salir con s**

***Nodo = estado inicial***

***Solución = solución***

***S = DFS\_prof\_iter(nodo, solución)***

# Vuelos con busqueda con profundidad iterativa

from arbol import Nodo

def DFS\_prof\_iter(nodo, solucion):

    for limite in range(0,100):

        visitados=[]

        sol = buscar\_solucion\_DFS\_Rec(nodo, solucion, visitados, limite)

        if sol != None:

           return sol

def buscar\_solucion\_DFS\_Rec(nodo, solucion, visitados, limite):

    if limite > 0:

        visitados.append(nodo)

        if nodo.get\_datos() == solucion:

            return nodo

        else:

            # expandir nodos hijo (ciudades con conexion)

            dato\_nodo = nodo.get\_datos()

            lista\_hijos =[]

            for un\_hijo in conexiones[dato\_nodo]:

                hijo = Nodo(un\_hijo)

                if not hijo.en\_lista(visitados):

                    lista\_hijos.append(hijo)

            nodo.set\_hijos(lista\_hijos)

            for nodo\_hijo in nodo.get\_hijos():

                if not nodo\_hijo.get\_datos() in visitados:

                    # llamada recursiva

                    sol = buscar\_solucion\_DFS\_Rec(nodo\_hijo, solucion, visitados, limite-1)

                    if sol != None:

                        return sol

        return None

if \_\_name\_\_=="\_\_main\_\_":

    conexiones = {

        'EDO.MEX':{'QRO' , 'SLP', 'SONORA'},

        'PUEBLA':{'HIDALGO', 'SLP'},

        'CDMX':{'MICHOACAN'},

        'MICHOACAN':{'SONORA'},

        'SLP':{'QRO', 'PUEBLA', 'EDO.MEX', 'SONORA', 'GUADALAJARA'},

        'QRO':{'EDO.MEX', 'SLP'},

        'HIDALGO':{'PUEBLA', 'GUADALAJARA', 'SONORA'},

        'GUADALAJARA':{'HIDALGO', 'SLP'},

        'MONTERREY':{'SONORA'},

        'SONORA':{'MONTERREY', 'HIDALGO', 'SLP', 'EDO.MEX', 'MICHOACAN'}

    }

    estado\_inicial = 'EDO.MEX'

    solucion = 'HIDALGO'

    nodo\_inicial = Nodo(estado\_inicial)

    nodo = DFS\_prof\_iter(nodo\_inicial, solucion)

    # Mostrar el resultado

    if nodo != None:

        resultado=[]

        while nodo.get\_padre() != None:

            resultado.append(nodo.get\_datos())

            nodo = nodo.get\_padre()

        resultado.append(estado\_inicial)

        resultado.reverse()

        print(resultado)

    else:

        print ("solucion no encontrada")

Aunque parezca un proceso pesado y costoso, en realidad no lo es tanto. Hay que tener en cuenta la operación más costosa es generar los nodos más profundos (hojas). En comparación, generar los niveles anteriores, es un proceso liviano. Además, si se encuentra la solución en el nivel **d** del árbol, los nodos de este nivel solo se habrán generado una vez, los del nivel **d-1** dos veces y así sucesivamente. Es decir, el numero de nodos total generados en un árbol con factor de ramificación **b** y profundidad **d** será:

Numero de nodos = b(d) +(d-1) + (d-2) +……

Por lo que su complejidad es **O (),** que es exactamente la misma que la de la búsqueda en amplitud. Es por ello, que si no se conoce la profundidad a la que se encuentra la solución y el espacio de búsqueda grande, este algoritmo será preferible a las otras búsquedas.

**°°° BÚSQUEDA DE COSTO UNIFORME °°°**

Los métodos de búsqueda analizados hasta ahora nos permiten enfrentarnos a un espectro de problemas relativamente amplio, pero no todos, por ejemplo, el problema de las rutas por carretera, con los algoritmos que ya se conocen, se puede abordar el problema, hasta cierto punto. Siempre se podrá encontrar una solución, pero como se sabe, la búsqueda en amplitud no es óptima, y es este precisamente uno de los requerimientos del problema.

¿Cómo obtener el camino optimo con menor distancia?

**La búsqueda de costo uniforme o Uniform Cost Search (UCS),** permite realizar una búsqueda en el espacio de estados, teniendo en cuenta un nuevo factor de **“costo”.**

En esta búsqueda se asigna a cada nodo un costo. Se define la función **g(ñ)** como el costo de recorrer el camino que va desde el nodo raíz al nodo que se está evaluando. Por lo tanto, el costo del camino será la suma de los costos de cada nodo recorrido.

Árbol de costos asociados (Ponderado)

C=COSTO

C= 15

C= 35

C= 25

C= 33

C= 12

C= 22

El árbol anterior muestra un árbol de estados donde se indica el costo asociado a cada nodo. Algunos ejemplos de función de costo **g(ñ)** son:

g(B) = 35 + 22 = 57

g(D) = 15

g(F) = 15 + 25 = 40

La búsqueda de costo uniforme utiliza una cola con prioridad en vez de una cola FIFO o una pila LIFO, para almacenar los nodos frontera. Una cola con prioridad, es una cola que se mantiene ordenada, en este caso la cola se ordena según el costo de cada nodo. También es posible usar una cola normal y ordenarla en cada paso, o alternativamente extraer el menor de los valores de la cola.

A continuación, se presenta el seudocódigo de búsqueda de costo uniforme:

**Pseudocódigo de búsqueda de costo uniforme**

**Nodo\_inicial=estado inicial**

**Nodos\_frontera=Cola con prioridad**

**Nodos visitados =Lista**

**Almacenar nodo inicial en nodos-frontera**

**Mientras nodos\_frontera no este vacío:**

**Ordenar la lista de nodos\_frontera según el costo**

**Nodo\_actual=extraer el primer nodo de nodos frontera**

**Si nodo\_actual == solución:**

**Salir con la solución**

**Introducir nodo\_actual en nodos\_visitados**

**Por cada operador:**

**Nodo\_hijo = operador(nodo\_actual)**

**Si nodo\_hijo no está en visitados:**

**Si nodo\_hijo no está en nodos\_frontera:**

**Si el costo de nodo\_hijo<nodo en nodos\_frontera:**

**Sustituir nodo\_hijo en nodos\_frontera**

**Si no:**

**Introducir nodo\_hijo en nodos\_frontera**

Por lo tanto, cada vez que se selecciona un nodo para expandir, nos quedamos con aquel que tiene menor costo.

La búsqueda de costo uniforme es óptima en todo caso, ya que siempre se toma el nodo de menor costo acumulado de la lista de nodos frontera. Cuando se encuentra un nodo solución se está seguro de que no existe una solución de menor costo, al no haberla encontrado anteriormente. La búsqueda será completa siempre que la profundidad del árbol sea finita y los costos sean positivos.

La complejidad, sin embargo, no están obvia. Hay que tener en cuenta que no se puede caracterizar en función de la profundidad del árbol, ya que no se sabe a priori porque ramas descenderá ni a que profundidad. El factor de ramificación por sí solo tampoco permite encontrar una medida de la complejidad de la búsqueda. En este caso se intentará acotar a partir del costo de la solución.

Suponiendo que la solución óptima se encuentra en el Nodo N, que los costos de todos los nodos son iguales o superiores a un valor C y que el factor de ramificación es b, la complejidad espacial y temporal del algoritmo es **O(b1+(g(N)/c)))**. La búsqueda uniforme en el que todos los costos son iguales; sin embargo. En condiciones no favorables, el costo puede llegar ser bastante superior al de la búsqueda en amplitud **O (bd**).

Retomando el problema del viaje por carretera, se aplicará este algoritmo con un ejemplo. El objetivo es encontrar el camino más corto desde EDO.MEX a Hidalgo. Se analizará cada paso y como el algoritmo selecciona los nodos. Se indicará el costo acumulado. En la lista de nodos frontera estará en negritas, aquellos nodos que se añaden nuevos o que cambien su valor **g(n).**

**Paso 1.**

Nodos frontera: EDO.MEX (0)

Seleccionar: EDO.MEX (0)

Hijos no visitados: SLP (513), CDMX (125)

**Paso 2.**

Nodos frontera: CDMX (125), SLP (513)

Seleccionar: CDMX (125)

Hijos no visitados: SLP (513), MICHOACAN (616)

**Paso 3.**

Nodos frontera: MICHOACAN (616)

Seleccionar: SLP (513)

Hijos no visitados: MICHOACAN (616), SONORA (1116), MONTERREY (826), GUADALAJARA (950), HIDALGO (1112), QRO (716), PUEBLA (1027)

**Paso 4.**

Nodos frontera: MICHOACAN (616), QRO (716), MONTERREY (826), GUADALAJARA (950), HIDALGO (1112), SONORA (1116), PUEBLA (1027).

Seleccionar: MICHOACAN (616).

Hijos no visitados: SONORA (962), MONTERREY (925).

**Paso 5.**

Nodos frontera: QRO (716), MONTERREY (826), GUADALAJARA (950), SONORA (962), HIDALGO (1112), PUEBLA (1027).

Seleccionar: QRO (716).

Hijos no visitados: HIDALGO (1106).

**Paso 6.**

Nodos frontera: MONTERREY (826), GUADALAJARA (950), SONORA (962), HIDALGO (1106), PUEBLA (1027).

Seleccionar: MONTERREY (826).

Hijos no visitados: SONORA (1122), QRO (1220).

**Paso 7.**

Nodos frontera: GUADALAJARA (950), SONORA (962), HIDALGO (1106), PUEBLA (1027).

Seleccionar: GUADALAJARA (950).

Hijos no visitados: NINGUNO.

**Paso 8.**

Nodos frontera: SONORA (962), HIDALGO (1106), PUEBLA (1027).

Seleccionar: SONORA (962).

Hijos no visitados: NINGUNO.

**Paso 9.**

Nodos frontera: HIDALGO (1106), PUEBLA (1027).

Seleccionar: HIDALGO (1106).

Hijos no visitados: NINGUNO.

En este punto se ha encontrado el nodo objetivo y por lo tanto la solución que se buscaba con un costo de 1,106 kilómetros. Se ha necesitado de 9 pasos, para encontrar la solución óptima.

Se tiene una posible implementación del algoritmo en Python. Por simplicidad no se ha utilizado ninguna biblioteca adicional para poder usar una cola con prioridad (que no incluye Python en su instalación por defecto). En su defecto, se podría usar una función sorted(), para ordenar la lista de nodos frontera. El problema con el que se enfrenta es que sorted() no sabe cómo ordenar objetos de la clase Nodo (se quiere ordenar la lista por el costo acumulado g(n)). Afortunadamente, esta función permite especificar como parámetro una función externa para indicarle como hacer la comparación.

***Nodos\_frontera = sorted(nodos\_frontera, cmp = compara)***

Aquí se le indica que utilice la función compara (x, y) para decidir como ordenar los nodos. Esta función recibe dos objetos x e y para ser comparados, se espera que devuelva un valor negativo si x < y, un valor positivo si x > y, y el valor 0 si x=y

Guadalajara

394

423

355

491

513

514

309

346

Querétaro

Hidalgo

Monterrey

296

437

599

Sonora

313

390

SLP

603

203

Michoacán

Puebla

CDMX

125

Edo.Méx

**Mapa de Carreteras**

CODIGO DE CARRETERA (UCS)

# Viaje por carretera con búsqueda de coste uniforme

import functools

from arbol import Nodo

def compara(x, y):

    return x.get\_costo() - y.get\_costo()

def buscar\_solucion\_UCS(conexiones, estado\_inicial, solucion):

    solucionado = False

    nodos\_visitados = []

    nodos\_frontera = []

    nodo\_inicial = Nodo(estado\_inicial)

    nodo\_inicial.set\_costo(0)

    nodos\_frontera.append(nodo\_inicial)

    while(not solucionado) and len(nodos\_frontera) != 0:

        # ordenar la lista de nodos frontera

        nodos\_frontera = sorted(nodos\_frontera, key=functools.cmp\_to\_key(compara))

        nodo = nodos\_frontera[0]

        # extraer nodo y añadirlo a visitados

        nodos\_visitados.append(nodos\_frontera.pop(0))

        if nodo.get\_datos() == solucion:

            # solución encontrada

            solucionado = True

            return nodo

        else:

            # Expandir nodos hijo (ciudades con conexión)

            dato\_nodo = nodo.get\_datos()

            lista\_hijos = []

            for un\_hijo in conexiones[dato\_nodo]:

                hijo = Nodo(un\_hijo)

                costo = conexiones[dato\_nodo][un\_hijo]

                hijo.set\_costo(nodo.get\_costo() + costo)

                lista\_hijos.append(hijo)

                if not hijo.en\_lista(nodos\_visitados) :

                    # si está en la lista lo sustituimos con

                    # el nuevo valor de coste si es menor

                    if hijo.en\_lista(nodos\_frontera):

                        for n in nodos\_frontera:

                            if n.igual(hijo) and n.get\_costo() > hijo.get\_costo():

                                nodos\_frontera.remove(n)

                                nodos\_frontera.append(hijo)

                    else:

                        nodos\_frontera.append(hijo)

                nodo.set\_hijos(lista\_hijos)

if \_\_name\_\_ == '\_\_main\_\_':

    conexiones = {

        'EDO.MEX':{'SLP' :513, 'CDMX' :125},

        'PUEBLA':{'SLP': 514},

        'CDMX':{'MICHOACAN': 491, 'SLP' :423,},

        'SLP':{'QRO': 203, 'PUEBLA': 514, 'EDO.MEX': 513, 'SONORA': 603, 'GUADALAJARA': 437, 'HIDALGO': 599, 'CDMX': 423, 'MICHOACAN': 355, 'MONTERREY': 313},

        'QRO':{'HIDALGO':390, 'SLP':203},

        'HIDALGO':{'QRO': 390, 'SLP': 599},

        'GUADALAJARA':{'SLP': 437, 'MONTERREY': 394},

        'MONTERREY':{'SONORA': 296, 'GUADALAJARA': 394,'SLP': 313, 'MICHOACAN': 309},

        'SONORA':{'MONTERREY': 296, 'SLP': 603, 'MICHOACAN': 346},

        'MICHOACAN':{'SLP': 355, 'CDMX' : 491, 'SONORA': 346, 'MONTERREY': 309 }

    }

estado\_inicial = 'EDO.MEX'

solucion = 'HIDALGO'

nodo\_solucion = buscar\_solucion\_UCS(conexiones, estado\_inicial, solucion)

#mostrar resultado

resultado = []

nodo= nodo\_solucion

while nodo.get\_padre() != None:

    resultado.append(nodo.get\_datos())

    nodo = nodo.get\_padre()

resultado.append(estado\_inicial)

resultado.reverse()

print(resultado)

print('Coste: ' + str(nodo\_solucion.get\_costo()))

**Resultado:**

['EDO.MEX', 'SLP', 'QRO', 'HIDALGO']

Coste: 1106

**°°° EL ALGORITMO DE DIJKSTRA °°°**

El algoritmo de **DIJKTRA** es un algoritmo constructivo voraz capaz de encontrar el camino más corto desde el nodo de un grafo ponderado el resto de nodos. Puede ser usado en multitud de escenarios. Se aplica exitosamente en algoritmos de enrutamiento de redes y telefonía, aplicaciones de cálculo de rutas etc.

El algoritmo de **Dijkstra** se va etiquetando cada nodo con la tupla [distancia\_acumulada,padre] donde distancia\_acumulada es la distancia mínima desde el nodo inicial y padre es el nodo predecesor en el camino mínimo que une el nodo inicial con el que se está calculando. La secuencia de pasos para ir etiquetando todos los nodos es:

1.- Seleccionar el nodo no visitado con menor distancia acumulada.

2.- Sumar la distancia acumulada con la distancia a los nodos adyacentes y se etiqueta con el de menor distancia acumulada.

3.- Se marca el nodo actual como visitado, regresar al paso 1.

Ejemplo: supóngase el siguiente grafo ponderado, que puede representar una serie de ciudades con su distancia, los nodos de una red o cualquier distancia que pueda moderarse con grafo.

**3+**

**2+**

**2+**

**2+**

**6+**

**2**

**6+**

**4+**

**3+**

**1**

***Grafo de ejemplo Dijkstra***

Se desea conocer la distancia mínima desde el nodo 1 hasta el resto de los nodos. Si cada nodo representa una ciudad, cada arista una conexión por carretera y el valor de la arista representa los kilómetros de distancia que separa cada ciudad, el algoritmo Dijkstra devolverá la distancia mínima desde la ciudad 1 al resto de ciudades.

A continuación, se presenta como se aplica el algoritmo.

**PASO 1: elegir el primer nodo(el nodo 1) y marcar con distancia 0 y sin padre**

**[0,-]**

**3+**

**2+**

**2+**

**2+**

**6+**

**2**

**6+**

**4+**

**3+**

**1**

**PASO 2: Se etiqueta los dos nodos adyacentes con la distancia y el nodo predecesor. También se marca el nodo 1 como visitado en rojo.**

**[6,1]**

**[0,-]**

**3+**

**2+**

**2+**

**2+**

**6+**

**2**

**6+**

**4+**

**3+**

**1**

**[3,1]**

**PASO 3: Seleccionar el nodo con menor distancia acumulada. En este caso el 2 tiene un peso acumulado de 3. Seleccionar los nodos adyacentes y se etiquetan con el peso acumulado (Peso del nodo actual más el peso de la arista del nodo adyacente). Como padre ahora coloca el nodo 2. Como nos podemos percatar el nodo3 tiene ahora dos etiquetas, así que nos quedaremos con la de menor peso. Se marca el nodo dos como visitado**

**[5,2]**

**[6,1]**

**[0,-]**

**3+**

**2+**

**2+**

**2+**

**6+**

**2**

**6+**

**4+**

**3+**

**1**

**[4,2]**

**[3,1]**

**[5,2]**

**PASO 4: Seleccionar el nodo 4 que es el de menor peso acumulado. En este caso ya no se tiene en cuenta el nodo 2, porque ya esta como nodo visitado, ni el nodo 3 por que el peso acumulado es menor que el nuevo, así que se etiqueta el nodo 5 y se marca el nodo 4 como visitado.**

**[8,4]**

**[5,2]**

**[10,4]**

**[0,-]**

**3+**

**2+**

**2+**

**2+**

**6+**

**2**

**6+**

**4+**

**3+**

**1**

**[4,2]**

**[3,1]**

**[5,2]**

**PASO 5: El siguiente nodo de menor peso acumulado es el 3, como el nodo 4 ya fue visitado, solo queda etiquetar el 5. En este caso, como el nuevo peso acumulado es menor, se re etiqueta con el nuevo peso y el nuevo padre. Se marca el nodo 3 como visitado.**

**[7,3]**

**[5,2]**

**[10,4]**

**[0,-]**

**3+**

**2+**

**2+**

**2+**

**6+**

**2**

**6+**

**4+**

**3+**

**1**

**[4,2]**

**[3,1]**

**[5,2]**

**PASO 6:** **Seleccionar el nodo 5 y se etiqueta los nodos adyacentes no visitados. Seguido se marcan como visitados.**

**[7,3]**

**[5,2]**

**[9,5]**

**[0,-]**

**3+**

**2+**

**2+**

**2+**

**6+**

**2**

**6+**

**4+**

**3+**

**1**

**[4,2]**

**[3,1]**

**[9,5]**

**[5,2]**

**PASO 7: Finalmente se selecciona el nodo 6 y como no mejora el peso acumulado hasta el nodo 7, no se cambia la etiqueta. Lo mismo ocurre con el 7, así que se marcan ambos, como visitados y se obtiene los siguientes pesos y rutas.**

**[7,3]**

**[5,2]**

**[9,5]**

**[0,-]**

**3+**

**2+**

**2+**

**2+**

**6+**

**2**

**6+**

**4+**

**3+**

**1**

**[4,2]**

**[3,1]**

**[9,5]**

**[5,2]**

Ahora es posible responder cualquier pregunta que se haga sobre cualquier grafo. Por ejemplo:

¿Cuál es la distancia (Peso) mínima desde el nodo 1 al nodo 7?

Directamente responder esta pregunta es muy sencillo, siendo la respuesta 9.

¿Cuál es el camino mínimo del nodo 1 al nodo 7? Solo hay que mirar quien es el padre del nodo 7 e ir recorriendo el camino hacia atrás, viendo quien es el padre de cada nodo hasta llegar al nodo 1. En este caso el camino mínimo es 1,2,3,5,7.

CODIGO DE DISKSTRA

import sys

def dijkstra(grafo, nodo\_inicial):

    etiquetas = {}

    visitados = []

    pendientes = [nodo\_inicial]

    nodo\_actual = nodo\_inicial

    # Nodo Inicial

    etiquetas[nodo\_actual] = [0,'']

    # Seleccionar el siguiente nodo de menor peso acumulado.

    while len(pendientes) > 0:

        nodo\_actual = nodo\_menor\_peso(etiquetas, visitados)

        visitados.append(nodo\_actual)

        # Obtener nodos adyacentes

        for adyacente, peso in grafo[nodo\_actual].items():

            if adyacente not in pendientes and adyacente not in visitados:

                pendientes.append(adyacente)

            nuevo\_peso = etiquetas[nodo\_actual][0] + grafo[nodo\_actual][adyacente]

            # Etiquetar

            if adyacente not in visitados:

                if adyacente not in etiquetas:

                    etiquetas[adyacente] = [nuevo\_peso, nodo\_actual]

                else:

                    if etiquetas[adyacente][0] > nuevo\_peso:

                        etiquetas[adyacente] = [nuevo\_peso, nodo\_actual]

        del pendientes[pendientes.index(nodo\_actual)]

    return etiquetas

def nodo\_menor\_peso(etiquetas, visitados):

    menor = sys.maxsize

    for nodo, etiqueta in etiquetas.items():

        if etiqueta[0] < menor and nodo not in visitados:

            menor = etiqueta[0]

            nodo\_menor = nodo

    return nodo\_menor

if \_\_name\_\_ == '\_\_main\_\_':

    grafo = {

        '1': {'3':6, '2':3},

        '2': {'4':1, '1':3, '3':2},

        '3': {'1':6, '2':2, '4':4, '5':2},

        '4': {'2':1, '3':4, '5':6},

        '5': {'3':2, '4':6, '6':2, '7':2},

        '6': {'5':2, '7':3},

        '7': {'5':2, '6':3}

    }

    etiquetas = dijkstra(grafo, '1')

    print(etiquetas)

**Resultado:**

{'1': [0, ''], '3': [5, '2'], '2': [3, '1'], '4': [4, '2'], '5': [7, '3'], '6': [9, '5'], '7': [9, '5']}

**°°° FUNCIÓN HEURÍSTICA °°°**

La función heurística, que llamaremos **h(n),** trata de guiar la busqueda para llegar de forma más rápida a la solución. A partir de la información disponible, la función heurística intenta estimar el coste del mejor camino (menor coste) desde el nodo n hasta el nodo objetivo. Por lo tanto, la elección de una buena función heurística es crucial para obtener una buena solución.

Mientras la función de coste **g(n)** se calcula a partir del camino que va desde el nodo raíz al nodo actual, la función heurística **h(n)** depende únicamente del nodo que se está analizando en ese momento.

**-Aplicar Función Heurística a código de Puzle Lineal–**

Necesitamos un criterio que nos indique si un movimiento nos acerca o nos aleja de la solución. Este criterio nos permitirá comparar el nodo padre con los nodos hijo. Esta comparacion la haremos en los términos de una función heurística **h(n)** que estimara la distancia desde un nodo cualquiera al nodo objetivo.

Para el caso del Puzle Lineal, una función que nos puede dar una estimación de la distancia al nodo objetivo es el número de piezas mal colocadas, es decir, más piezas bien colocadas indican que estamos mas cerca de la solución. De esta forma, un nodo hijo que tiene mas piezas descolocadas que su padre nos dice que el movimiento empeora la solución. Dicho de otra manera, el nodo hijo es de peor calidad que el nodo padre.

Alternativamente podríamos usar otras funciones heurísticas **h(n);** por ejemplo, podemos medir la calidad de un nodo por el número de movimientos que lo separan de un nodo objetivo en vez de por el número de piezas mal colocadas. Esta es probablemente una mejor heurística, pero por sencillez en la implementación nos quedaremos con la primera.

**-Puzle Lineal con Heurística–**

# Puzle lineal con heurística

from árbol import Nodo

def buscar\_solucion\_heuristica(nodo\_inicial, solucion, visitados):

    visitados.append(nodo\_inicial.get\_datos())

    if nodo\_inicial.get\_datos() == solucion:

        return nodo\_inicial

    else:

        # Expandir nodos sucesores (hijos)

        dato\_nodo = nodo\_inicial.get\_datos()

        hijo = [dato\_nodo[1], dato\_nodo[0], dato\_nodo[2], dato\_nodo[3]]

        hijo\_izquierdo = Nodo(hijo)

        hijo = [dato\_nodo[0], dato\_nodo[2], dato\_nodo[1], dato\_nodo[3]]

        hijo\_central = Nodo(hijo)

        hijo = [dato\_nodo[0], dato\_nodo[1], dato\_nodo[3], dato\_nodo[2]]

        hijo\_derecho = Nodo(hijo)

        nodo\_inicial.set\_hijos([hijo\_izquierdo, hijo\_central, hijo\_derecho])

        for nodo\_hijo in nodo\_inicial.get\_hijos():

             if not nodo\_hijo.get\_datos() in visitados and mejora(nodo\_inicial, nodo\_hijo):

                 # Llamada Recursiva

                sol = buscar\_solucion\_heuristica(nodo\_hijo, solucion, visitados)

                if sol != None:

                    return sol

        return None

def mejora(nodo\_padre, nodo\_hijo):

    calidad\_padre=0

    calidad\_hijo=0

    dato\_padre = nodo\_padre.get\_datos()

    dato\_hijo = nodo\_hijo.get\_datos()

    for n in range(1,len(dato\_padre)):

        if (dato\_padre[n]>dato\_padre[n-1]):

            calidad\_padre = calidad\_padre + 1

        if (dato\_hijo[n]>dato\_hijo[n-1]):

            calidad\_hijo = calidad\_hijo + 1

    if calidad\_hijo>=calidad\_padre:

        return True

    else:

        return False

if \_\_name\_\_ == "\_\_main\_\_":

    estado\_inicial = [4, 2, 3, 1]

    solucion = [1, 2, 3, 4]

    nodo\_solucion = None

    visitados = []

    nodo\_inicial = Nodo(estado\_inicial)

    nodo = buscar\_solucion\_heuristica(nodo\_inicial, solucion, visitados)

    # Mostrar Resultado

    resultado = []

    while nodo.get\_padre() != None:

        resultado.append(nodo.get\_datos())

        nodo = nodo.get\_padre()

    resultado.append(estado\_inicial)

    resultado.reverse()

    print(resultado)

**Resultado: [[4, 2, 3, 1], [2, 4, 3, 1], [2, 3, 4, 1], [2, 3, 1, 4], [2, 1, 3, 4], [1, 2, 3, 4]]**

La ejecución del programa nos ofrece una solución similar a la que encontramos con la búsqueda en amplitud, pero usando una búsqueda en profundidad y, por lo tanto, menos costosa en cuanto al uso de memoria.

Hemos utilizado una función llamada **mejora()** que indica si el nodo hijo mejora respecto al nodo padre en los términos de calidad que hemos definido más arriba. Es decir, es una función booleana que devuelve el valor verdadero si el nodo hijo tiene más piezas en el lugar correcto que el padre, o al menos las mismas.

**°°° BUSQUEDA CON VUELTA ATRÁS (BACKTRACKING) °°°**

Una técnica comúnmente usada para mejorar el rendimiento temporal de la búsqueda en profundidad usando una función heurística es la **Búsqueda con vuelta atrás o BACKTRACKING.**

Cuando exploramos una rama del árbol, hay situaciones que no nos van a llevar a una solución o en el mejor de los casos, empeorarán la solución que se está analizando en ese momento. Es decir, que el nodo hijo empeora la situación respecto al nodo padre y por lo tanto nos aleja de la solución en vez de acercarnos a ella. Si un movimiento nos lleva a una rama en la que sabemos que no llegaremos a una solución, decimos que el **camino (la solución)** que estamos analizando no es **completable**. Como no tiene sentido seguir explorando una rama que es no completable, lo que se debe de hacer es dejar de explorar y volver hacia atrás (de ahí el nombre del algoritmo).

***Pseudocódigo búsqueda con vuelta atrás***

**Función DFS\_backtracking(nodo, solucion, visitados) :**

**Añadir nodo a visitados**

**Si nodo == solucion:**

**Salir con solucion**

**Si no:**

**Por cada operador:**

**nodo\_hijo = operador(nodo)**

**Si nodo\_hijo no en visitados y es\_completable(nodo\_hijo):**

**s = DFS\_backtracking(nodo\_hijo, solucion, visitados)**

**Salir con s**

**nodo = estado inicial**

**visitados = Lista**

**solucion = solucion**

**s = DFS\_backtracking(nodo, solucion, visitados)**

Retomando el problema de la **PLE (revisar apartado en el manual),** donde se nos pedía maximizar una función que representaba el beneficio resultante de la fabricación de x1 camisetas y x2 pantalones.

Se define en primera instancia el beneficio según la siguiente ecuación.

**Beneficio = (precio de venta – costo de producción) \* unidades vendidas**

Como el beneficio total será la suma de los beneficios de ambas prendas, llegamos a la siguiente ecuación que representa la función de beneficio para este caso en concreto.

Con las siguientes Restricciones:

Ambas variables han de ser mayores o iguales a 0. No se puede fabricar un número negativo de prendas.

El objetivo es encontrar una asignación de valores para las variables x1 y x2 que, cumpliendo con las restricciones impuestas, hagan que la función f tomo el máximo valor posible. Recordando que se llegó a la conclusión de que x1 estaba acotado entre 0 y 50 y x2 entre 0 y 75.

Al explorar el árbol y encontrar asignaciones que hacen que se incumplan las restricciones, no tiene sentido seguir explorando la rama, por lo que se pasa a la siguiente.

A continuación, se demuestra cómo afrontar el problema de **PLE(Programación lineal entera)** usando técnicas de **BACKTRACKING.** En el ejemplo no se usará la clase NODO ni se construirá el árbol como lo hemos hecho en otros ejemplos, ya que, a diferencia de los otros problemas tratados, en este no importa los pasos intermedios necesarios para llegar a la solución. Solo el resultado final. Solo nos interesan los valores finales y óptimos de x1 y x2. Por lo que se realiza una busqueda en un árbol, pero se van a ir generando los nodos según se van necesitando.

Para nuestro problema, que es bastante sencillo ya que solo tiene dos variables, se necesita hacer una búsqueda en un árbol de solo dos niveles de profundidad. En el primer nivel (sin contar el nodo raíz) se van a generar los posibles valores de x1, es decir, de 0 a 50. En el segundo nivel, por cada posible valor de x1, se van a generar todos los posibles valores de x2 que son los valores entre 0 y 75. En total se examinaran 50 x 75 = 3750 nodos. Sin embargo, ¿es necesario examinar todos? Supongamos que x1 = 25 y recordemos que una de las restricciones era:

Como es mayor que 150, queda claro que esta asignación incumple la restricción sea cual sea el valor de x2. En este escenario no tiene sentido desplegar los 75 nodos hijo para la asignación de x2.

**……………........**

**……………........**

**X1**

n

0

n/2

**X2**

n

0

n

0

0

n

**……**

**……**

**……**

**Despliegue del árbol para el problema PLE**

En nuestro ejemplo tenemos solo dos variables y 3.750 nodos por examinar, pero cualquier problema mediano puede tener bastantes variables y un número de nodos mucho mayor, por lo que resulta muy interesante aplicar esta técnica.

**- Código de BACKTRACKING con problema de PLE –**

# PLE con Backtracking

def backtracking(variables, rango\_variables, optimo, profundidad):

    min = rango\_variables[profundidad][0]

    max = rango\_variables[profundidad][1]

    for v in range(min, max):

        variables[profundidad] = v

        if profundidad < len(variables) -1:

            # Es completable si no cumple ninguna restricción

            if es\_completable(variables):

                optimo = backtracking(variables[:], rango\_variables, optimo, profundidad+1)

        else:

            # Estamos en una hoja. Comprobamos solucion

            sol = evalua\_solucion(variables)

            if sol > evalua\_solucion(optimo) and es\_completable(variables):

                optimo = (variables[0], variables[1])

    return optimo

def evalua\_solucion(variables):

    x1 = variables[0]

    x2 = variables[1]

    val = (12 - 6) \* x1 + (8 - 4) \* x2

    return val

def es\_completable(variables):

    x1 = variables[0]

    x2 = variables[1]

    val1 = 7 \* x1 + 4 \* x2

    val2 = 6 \* x1 + 5 \* x2

    if val1 <= 150 and val2 <= 160:

        return True

    else:

        return False

if \_\_name\_\_ == '\_\_main\_\_':

    # Valores de las variables x1 y x2

    variables = [0, 0]

    # Rango de las variables x1 y x2

    rango\_variables = [(0, 51), (0, 76)]

    # Mejor solucion encontrada

    optimo = (0, 0)

    sol = backtracking(variables[:], rango\_variables, optimo, 0)

    print('Mejor Solucion:')

    print(str(sol[0]) + ' Pantalones')

    print(str(sol[1]) + ' Camisetas')

    print('Beneficio: ' + str(evalua\_solucion(sol)))

**Resultado: Mejor Solucion: 10 Pantalones, 20 Camisetas Beneficio: 140**

La ejecución del programa nos indica que la mejor opción es fabricar 10 pantalones y 20 camisetas, lo que nos dejará un beneficio de 140 euros.

**°°° ALGORITMO A\* (A estrella) °°°**

Utilizando el concepto de coste, la búsqueda de coste uniforme nos permite llegar a una solución óptima. Sin embargo, hay que pagar un precio por ello. En condiciones desfavorables, la complejidad temporal y espacial del algoritmo puede crecer bastante. El algoritmo A\* (A estrella) trata de paliar la situación intentando que la convergencia hasta la solución sea más rápida. Para ello, hace uso de una función heurística h(n) que trata de estimar el coste desde el nodo n hasta el nodo objetivo.

Recordemos que la búsqueda de coste uniforme hacía uso de la función g(n) que llamábamos coste (coste acumulado desde el nodo raíz hasta el nodo n), de esta forma, a cada nodo se le hacía corresponder un valor según la siguiente función de evaluación:

El algoritmo A\* se basa en el mismo concepto, pero usa la siguiente función f(n) como función de evaluación.

Es decir, la suma del coste acumulado desde el nodo raíz más el coste estimado hasta el nodo solución. En cada iteración del algoritmo iremos seleccionando aquellos nodos con mayor valor f(n). Visto de esta manera, la búsqueda de coste uniforme es un caso especial del algoritmo A\* en el que h(n) toma el valor cero.

h(n) es una estimación; es decir, tendremos que basarnos en la información de la que disponemos para hacer la estimación. La elección de una buena función heurística h(n) es muy importante para que el algoritmo llegue a una solución rápida en pocos pasos.

***Pseudocódigo algoritmo A\****

**nodo\_inicial = estado inicial**

**nodos\_frontera = Cola con prioridad**

**nodos\_visitados = Lista**

**almacenar nodo\_inicial en nodos\_frontera**

**mientras nodos\_frontera no vacío:**

**calcular el valor f(n)=g(n)+h(n) para cada nodo frontera**

**ordenar la lista de nodos\_frontera según f(n)**

**nodo\_actual = extraer el primer nodo de nodos\_frontera**

**si nodo\_actual == solución:**

**salir con solución**

**introducir nodo\_actual en nodos\_visitados**

**por cada operador:**

**nodo\_hijo = operador(nodo\_actual)**

**si nodo\_hijo no en nodos\_visitados:**

**si nodo\_hijo en nodos\_frontera:**

**si coste de nodo\_hijo < nodo en nodos\_frontera:**

**sustituir nodo\_hijo en nodos\_frontera**

**si no:**

**introducir nodo\_hijo en nodos\_frontera**

**-Aplicar Algoritmo A\* a código de Carreteras–**

Retomemos el ejemplo del cálculo de la mejor ruta para el viaje por carretera. Cuando analizamos este problema bajo el prisma de la búsqueda de coste uniforme, utilizamos como función de coste g(n) la distancia recorrida desde la ciudad inicial hasta la ciudad representada por el nodo frontera que se evaluaba en cada caso. ¿Qué función heurística h(n) podemos usar para este mismo problema? Parece que una buena estimación podría ser la distancia en línea recta desde el nodo que estamos analizando hasta el nodo de la ciudad de destino. Siempre y cuando podamos obtener este dato. Supongamos que disponemos de la siguiente tabla con las coordenadas geográficas de cada una de las ciudades.

|  |  |  |
| --- | --- | --- |
| Ciudad | Latitud | Longitud |
| Guadalajara | 43,28 N | 3.48 O |
| Hidalgo | 42.52 N | 8.33 O |
| Querétaro | 40.57 N | 5.40 O |
| San Luis Potosí | 40.24 N | 3.41 O |
| Monterrey | 41.39 N | 0.52 O |
| Sonora | 41.23 N | 2.11 E |
| Puebla | 37.23 N | 5.59 O |
| EDO.MEX | 36.43 N | 4.25 O |
| CDMX | 37.11 N | 3.35 O |
| Michoacán | 39.28 N | 0.22 O |

A partir de las coordenadas de las ciudades, podemos obtener su distancia en línea recta medida en kilómetros. Son las llamadas distancias geodésicas. En el ecuador los meridianos de longitud separados por un grado se encuentran a una distancia de 111,32km, mientras que en los polos los meridianos convergen. Cada grado de longitud y de latitud se divide en 60 minutos y cada minuto en 60 segundos. De este modo se puede asignar una localización precisa a cualquier lugar de la Tierra. Los detalles para el cálculo de distancias geodésicas escapan del ámbito de este libro, así que nos limitaremos a decir que para hacer el cálculo usaremos la siguiente función.

**def geodist(lat1, lon1, lat2, lon2):**

**grad\_rad = 0.01745329**

**rad\_grad = 57.29577951**

**longitud = long1-long2**

**val=(sin(lat1\*grad\_rad)\*sin(lat2\*grad\_rad))+(cos(lat1\*grad\_rad)\*cos(lat2\*grad\_rad)\*cos(longitud\*grad\_rad))**

**return (acos(val)\*rad\_grad)\*111.32**

La función **geodist()** toma como parámetros la latitud y longitud de las dos coordenadas sobre las que queremos conocer la distancia y nos devuelve la distancia en kilómetros. La función devuelve un número real, por lo que habrá que hacer un casting a entero para poder usar su valor con la función **sorted().**

**-Viale por Carretera con A\*–**

# Viaje por carretera con Búsqueda A\*

import functools

from arbol import Nodo

from math import sin, cos, acos

def compara(x, y):

    # g(n) + h(n) para ciudad x

    lat1 = coord[x.get\_datos()][0]

    lon1 = coord[x.get\_datos()][1]

    lat2 = coord[solucion][0]

    lon2 = coord[solucion][1]

    d = int(geodist(lat1, lon1, lat2, lon2))

    c1 = x.get\_coste() + d

    # g(n) + h(n) para ciudad y

    lat1 = coord[y.get\_datos()][0]

    lon1 = coord[y.get\_datos()][1]

    lat2 = coord[solucion][0]

    lon2 = coord[solucion][1]

    d = int(geodist(lat1, lon1, lat2, lon2))

    c2 = y.get\_coste() + d

    return c1 - c2

def geodist(lat1, lon1, lat2, lon2):

    grad\_rad = 0.01745329

    rad\_grad = 57.29577951

    longitud = lon1 - lon2

    val = (sin(lat1 \* grad\_rad) \* sin(lat2 \* grad\_rad)) + (cos(lat1 \* grad\_rad) \* cos(lat2 \* grad\_rad) \* cos(longitud \* grad\_rad))

    return (acos(val) \* rad\_grad) \* 111.32

def buscar\_solucion\_UCS(conexiones, estado\_inicial, solucion):

    solucionado = False

    nodos\_visitados = []

    nodos\_frontera = []

    nodo\_inicial = Nodo(estado\_inicial)

    nodo\_inicial.set\_coste(0)

    nodos\_frontera.append(nodo\_inicial)

    while (not solucionado) and len(nodos\_frontera) != 0:

        # Ordenar la lista de nodos Frontera

        nodos\_frontera = sorted(nodos\_frontera, key=functools.cmp\_to\_key(compara))

        nodo = nodos\_frontera[0]

        # Extraer nodo y añadirlo a nodos\_visitados

        nodos\_visitados.append(nodos\_frontera.pop(0))

        if nodo.get\_datos() == solucion:

            # Solucion Encontrada

            solucionado = True

            return nodo

        else:

            # Expandir nodos hijo (ciudades con conexion)

            dato\_nodo = nodo.get\_datos()

            lista\_hijos = []

            for un\_hijo in conexiones[dato\_nodo]:

                hijo = Nodo(un\_hijo)

                # Calculo g(n): Coste acumulado

                coste = conexiones[dato\_nodo][un\_hijo]

                hijo.set\_coste(nodo.get\_coste() + coste)

                lista\_hijos.append(hijo)

            if not hijo.en\_lista(nodos\_visitados):

                # Si esta en lista lo sustituimos con el

                # nuevo valor de coste si es menor

                if hijo.en\_lista(nodos\_frontera):

                    for n in nodos\_frontera:

                        if n.igual(hijo) and n.get\_coste() > hijo.get\_coste():

                            nodos\_frontera.remove(n)

                            nodos\_frontera.append(hijo)

                else:

                    nodos\_frontera.append(hijo)

        nodo.set\_hijos(lista\_hijos)

if \_\_name\_\_ == '\_\_main\_\_':

    conexiones = {

        'edomex':{'cdmx':125, 'slp':513},

        'puebla':{'slp':514},

        'cdmx':{'edomex':125, 'slp':423, 'michoacan':491},

        'michoacan':{'cdmx':491, 'slp':356, 'monterrey':309, \

        'sonora':346},

        'slp':{'queretaro':203, 'puebla':514, 'edomex':513, \

        'cdmx':423,'sonora':603, 'guadalajara':437, 'michoacan':356,\

        'monterrey':313, 'guadalajara':437, 'hidalgo':599},

        'queretaro':{'hidalgo':390, 'slp':203},

        'hidalgo':{'queretaro':390, 'slp':599},

        'guadalajara':{'slp':437, 'monterrey':394},

        'monterrey':{'sonora':296, 'michoacan':309, 'slp':313},

        'sonora':{'monterrey':296, 'slp':603, 'michoacan':346}

    }

    coord = {

        'edomex':(36.43, -4.24),

        'puebla':(37.23, -5.59),

        'cdmx':(37.11, -3.35),

        'michoacan':(39.28, -0.22),

        'slp':(40.24, -3.41),

        'queretaro':(40.57, -5.40),

        'hidalgo':(42.52, -8.33),

        'guadalajara':(43.28, -3.48),

        'monterrey':(41.39, -0.52),

        'sonora':(41.23, +2.11)

    }

    estado\_inicial='edomex'

    solucion='hidalgo'

    nodo\_solucion = buscar\_solucion\_UCS(conexiones, estado\_inicial, solucion)

    # Mostrar resultado

    resultado=[]

    nodo=nodo\_solucion

    while nodo.get\_padre() != None:

        resultado.append(nodo.get\_datos())

        nodo = nodo.get\_padre()

    resultado.append(estado\_inicial)

    resultado.reverse()

    print (resultado)

    print ('Coste: ' + str(nodo\_solucion.get\_coste()))

**Resultado:**

**['edomex', 'slp', 'hidalgo']**

**Coste: 1112**

Recordemos que la búsqueda de coste uniforme necesitó 9 pasos para llegar a una solución. Tal y como hicimos en el capítulo anterior, vamos a hacer un seguimiento paso por paso del algoritmo para poder comparar su efectividad. Entre paréntesis se indica el coste g(n) más el valor de h(n) correspondiente a la distancia en línea recta. Se adjunta también la tabla de distancias en línea recta desde el nodo objetivo (Hidalgo) hasta el resto de ciudades.

|  |  |
| --- | --- |
| Ciudad | Distancia en línea recta hasta Hidalgo |
| Monterrey | 658 |
| Hidalgo | 0 |
| Guadalajara | 404 |
| EDOMEX | 763 |
| Queretaro | 326 |
| San Luis Potosí | 482 |
| Michoacan | 771 |
| Puebla | 633 |
| Sonora | 876 |
| CDMX | 737 |

**Paso 1.**

Nodos frontera: EDO.MEX (0 + 763 = 763)

Seleccionar: EDO. MEX (0 + 763 = 763)

Hijos no visitados: SLP (513 + 482=995), CDMX (125 + 737=862)

**Paso 2.**

Nodos frontera: CDMX (125 + 737=862), SLP (513 + 4820995)

Seleccionar: CDMX (125 + 737=862),

Hijos no visitados: MICHOACAN (616 + 771 =1387)

**Paso 3.**

Nodos frontera: SLP (513 + 482=995), MICHOACAN (616 + 771 =1387)

Seleccionar: SLP (513 + 482=995)

Hijos no visitados: MICHOACAN (616 + 771 =1387), SONORA (1116 + 076 = 1992),

MONTERREY (826 + 658 = 1484), GUADALAJARA (950 + 404 =1354), HIDALGO (1112 + 0 = 1112),

QRO (716 +326 = 1042), PUEBLA (1027 +633 = 1660)

**Paso 4.**

Nodos frontera: QRO (716 +326 = 1042), HIDALGO (1112 + 0 = 1112), GUADALAJARA (950 + 404 =1354), MICHOACAN (616 + 771 =1387), MONTERREY (826 + 658 = 1484), PUEBLA (1027 +633 = 1660), SONORA (1116 + 076 = 1992)

Seleccionar: QRO (716 +326 = 1042)

Hijos no visitados: HIDALGO (1112 + 0 = 1112)

**Paso 5.**

Nodos frontera: HIDALGO (1112 + 0 = 1112), GUADALAJARA (950 + 404 =1354), MICHOACAN (616 + 771 =1387), MONTERREY (826 + 658 = 1484), ), PUEBLA (1027 +633 = 1660), SONORA (1116 + 076 = 1992)

Seleccionar: HIDALGO (1112 + 0 = 1112)

Con esto hemos llegado al nodo objetivo en solo 5 pasos contra los 9 de la búsqueda de coste uniforme. Una mejora de casi el 50%. Con árboles más grandes la mejora es incluso mayor.

Al seleccionar una función heurística h(n) hay que tener en cuenta que se deben cumplir las siguientes condiciones para que el algoritmo A\* funcione correctamente.

- h(n) ≥ 0 para todo n.

- h(n) = 0 si n es nodo objetivo.

- h(n) Ha de ser admisible.

Una función heurística admisible es aquella que nunca sobreestime el coste para llegar al nodo objetivo. Dicho de otra forma, la estimación que haga h(n) siempre ha de estar por debajo del coste real para llegar de n al nodo objetivo. En el ejemplo del viaje por carretera está claro que nunca vamos a sobreestimar el coste, ya que no hay un camino más corto que la línea recta. En este sentido, decimos que una heurística admisible es optimista, ya que su valor siempre está por debajo del valor del coste real.

Para dejar claro el concepto vamos a ver otro ejemplo de heurística admisible: Una empresa fabricante de vehículos monta 4 tipos de ruedas en sus modelos de vehículos. Los tipos son: Tipo T para vehículos de gama baja, tipo H para vehículos de gama media, tipo V para vehículos de gama alta y tipo W para vehículos de lujo. A su vez, el fabricante trabaja con cuatro distribuidores de rueda distintos. Los cuatro distribuidores pueden servir los cuatro tipos de rueda, pero cada uno tiene precios diferentes. Por razones comerciales interesa trabajar con las cuatro distribuidoras, por lo que se ha decidido comprar un tipo de rueda a cada fabricante. Los precios en euros por fabricante y tipo de rueda son los siguientes:

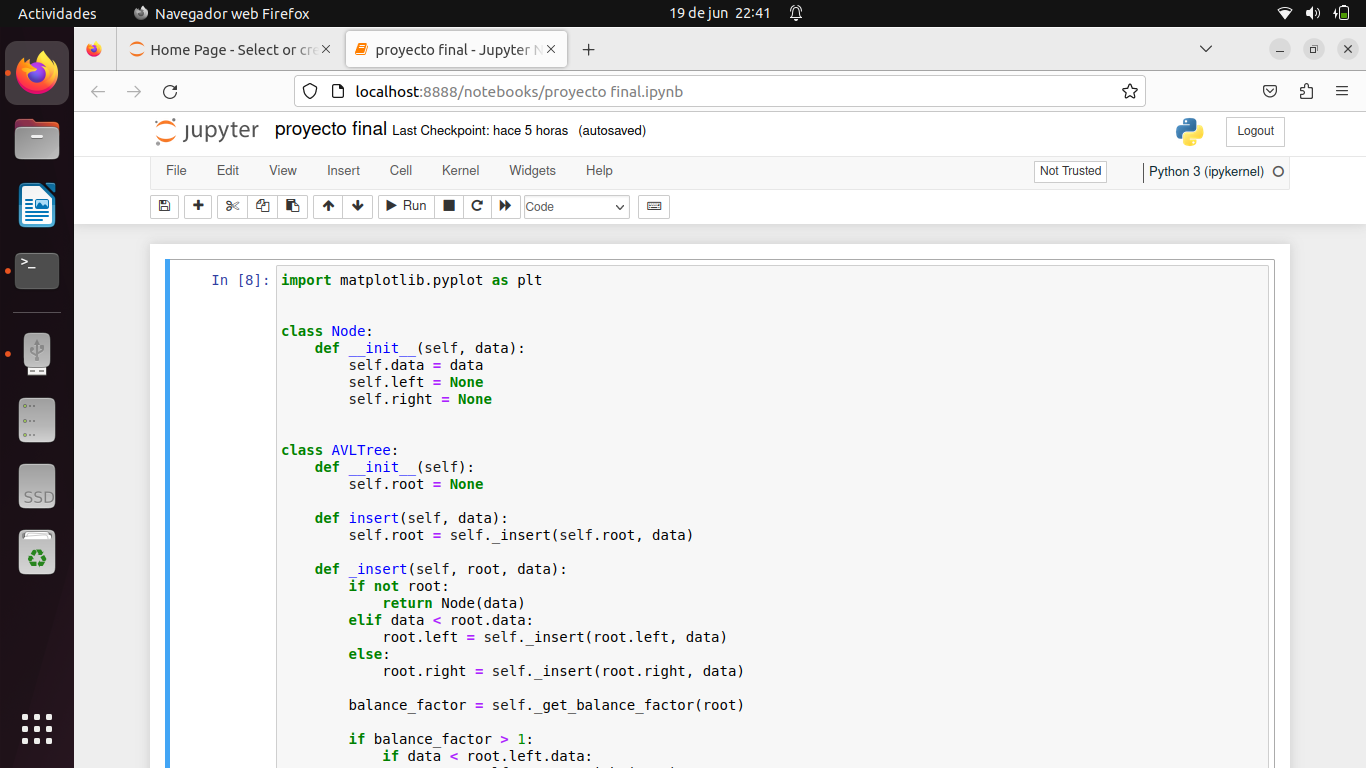
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Tipo T | Tipo H | Tipo V | Tipo W |
| Empresa 1 | 20 | 30 | 20 | 40 |
| Empresa 2 | 50 | 50 | 40 | 50 |
| Empresa 3 | 60 | 55 | 50 | 60 |
| Empresa 4 | 100 | 80 | 60 | 70 |

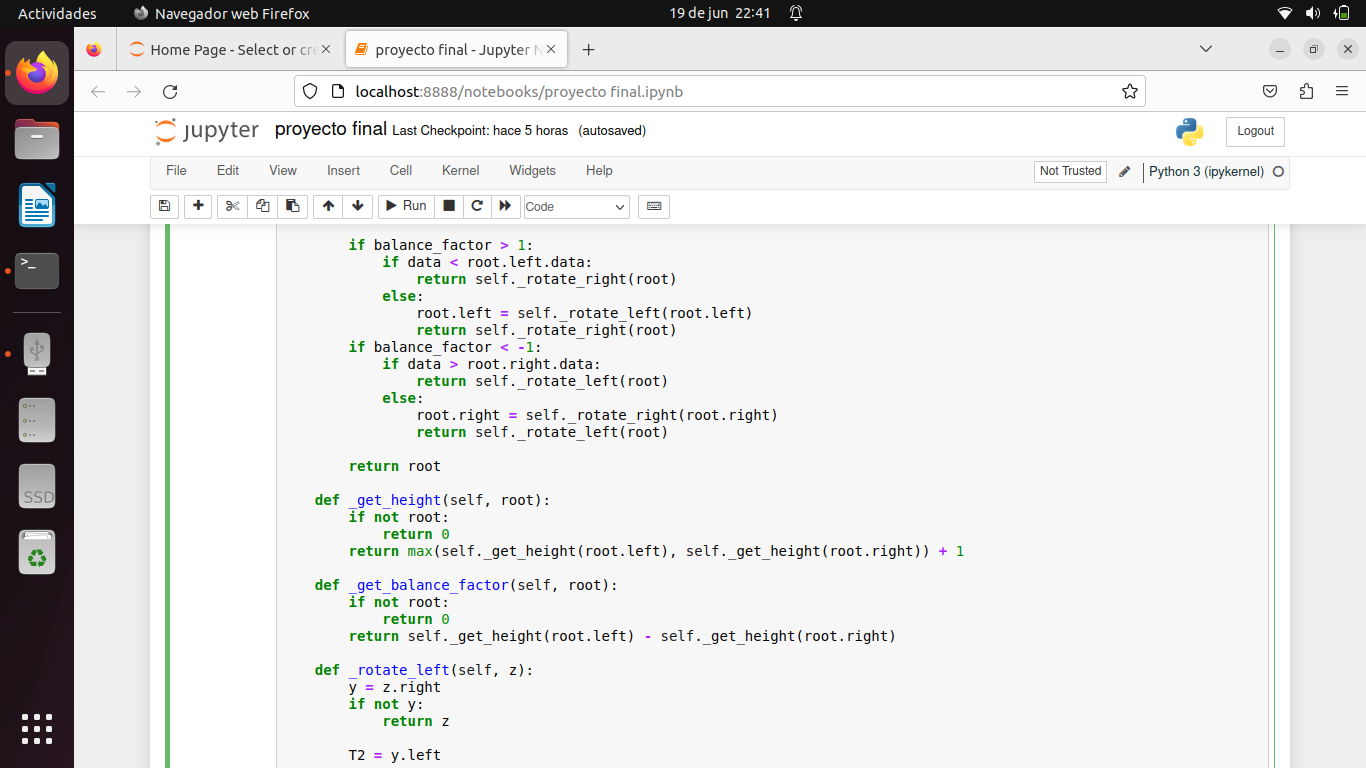
La empresa fabricante nos ha pedido que hagamos un estudio para seleccionar qué rueda comprar a cada fabricante de forma que el precio sea el más ventajoso posible. El problema puede atacarse con cualquiera de los tipos de búsqueda que hemos analizado, ya que las soluciones pueden representarse en forma de árbol. En el primer nivel tendríamos cuatro nodos representando al fabricante seleccionado para la rueda tipo 1. En el siguiente nivel tendríamos 3 nodos por cada uno de los cuatro nodos del nivel 1 representando al fabricante para el tipo de rueda 2 y así sucesivamente. En este escenario, la elección de la función g(n) es directa. El coste en euros de cada tipo de rueda. Sin embargo, la elección de una función h(n) que además sea una heurística admisible no es tan evidente. Una posible función heurística admisible podría ser la siguiente: Dado un estado n, la función h(n) será la suma de los precios más bajos de los tipos de rueda aún no asignados, teniendo solo en cuenta aquellas empresas no adjudicadas aún. Por ejemplo, supongamos que hemos adjudicado el tipo T a la empresa 1 y el nodo actual n evalúa la posibilidad de asignar el tipo H a la empresa 2. Las funciones g(n) y h(n) serían:

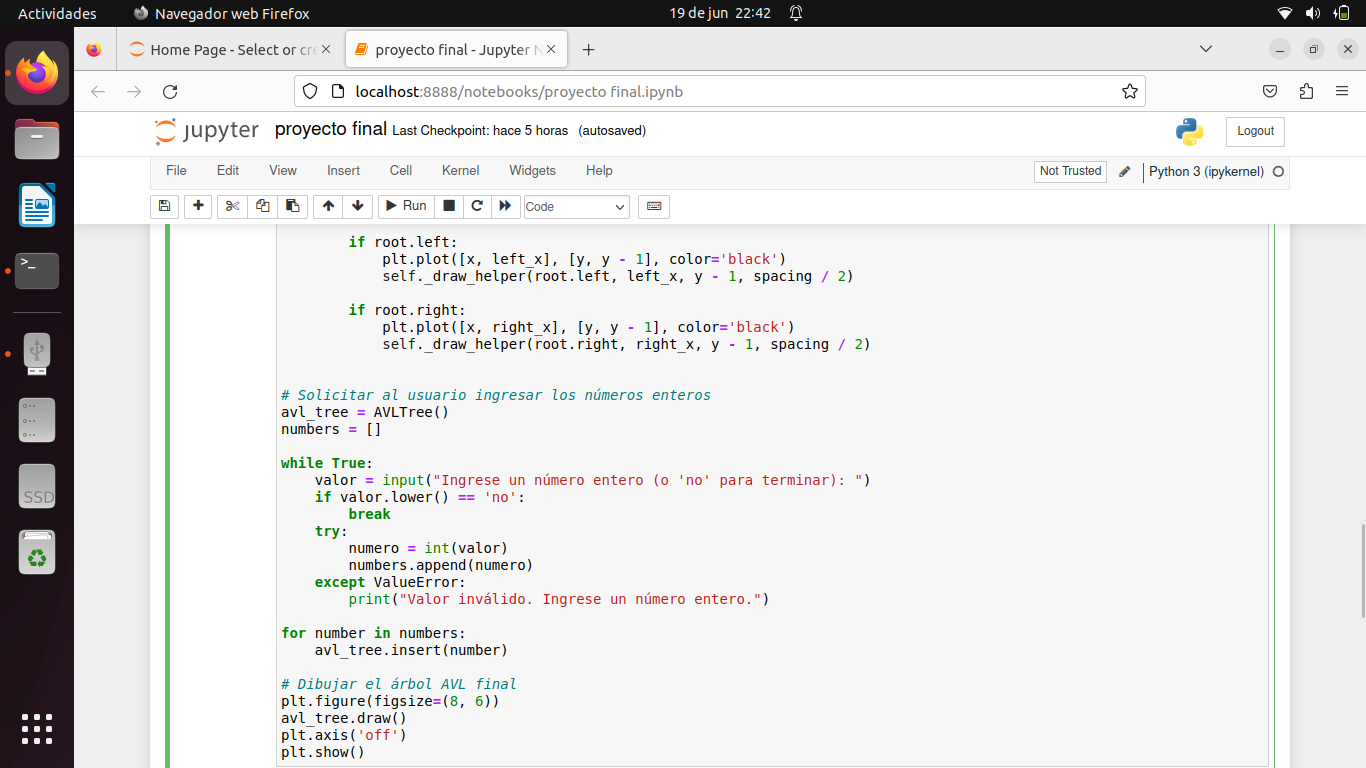
El valor 50 es el correspondiente al precio de la rueda tipo V de la empresa 3 y el valor 60 es el de la rueda tipo W también de la empresa 3. Como nos queda por escoger un precio para las ruedas tipo V y W de entre las empresas 3 y 4, queda claro que, sea cual sea la selección que hagamos, el precio será igual o superior a 110. Por lo tanto, la función h(n) es admisible.

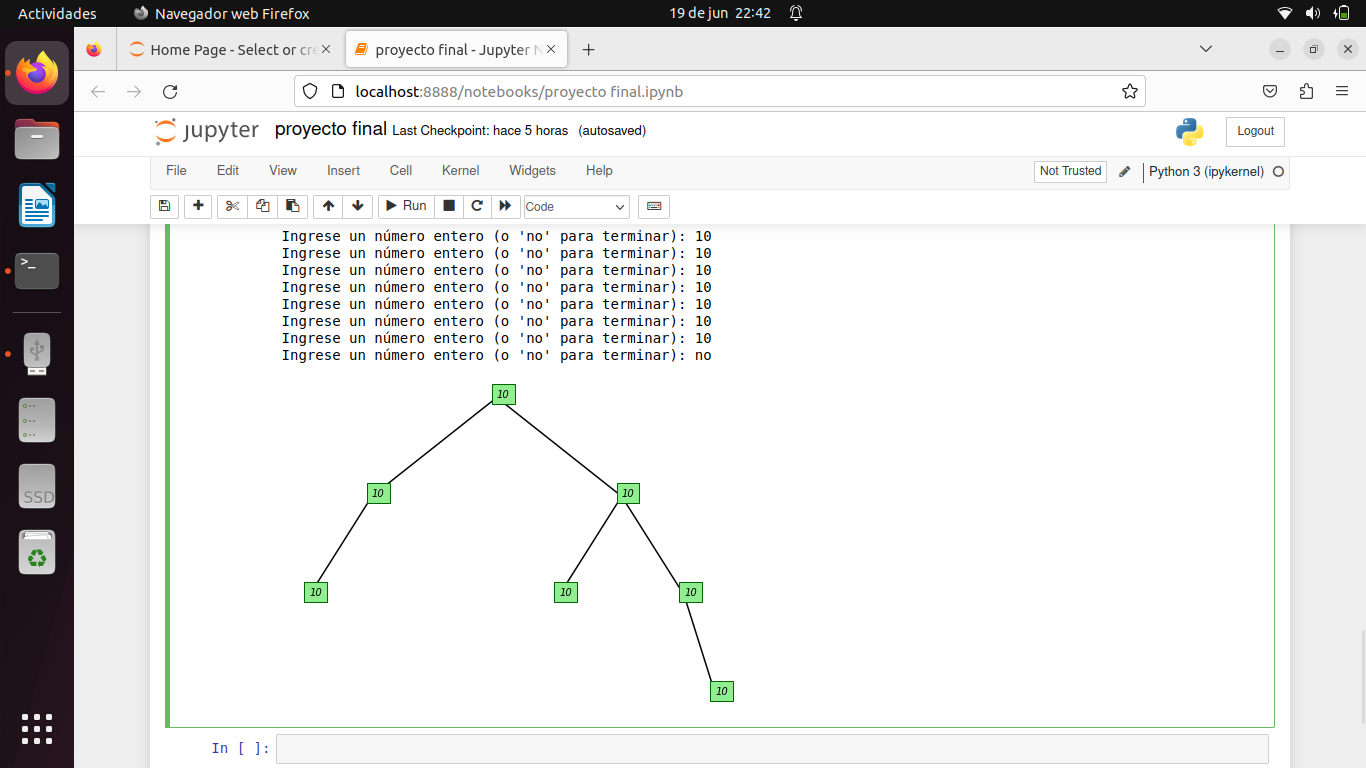
**°°° ARBOL AVL PROYECTO FINAL(EN PYTHON)°°°**

* Balancear un numero n de nodos, el cual se desarrolló en el lenguaje Python, utilizando el siguiente código. Se anexa evidencia de resultados de ejecución y código implementado.









**-Arbol AVL balanceado–**

import matplotlib.pyplot as plt

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

        self.right = None

class AVLTree:

    def \_\_init\_\_(self):

        self.root = None

    def insert(self, data):

        self.root = self.\_insert(self.root, data)

    def \_insert(self, root, data):

        if not root:

            return Node(data)

        elif data < root.data:

            root.left = self.\_insert(root.left, data)

        else:

            root.right = self.\_insert(root.right, data)

        balance\_factor = self.\_get\_balance\_factor(root)

        if balance\_factor > 1:

            if data < root.left.data:

                return self.\_rotate\_right(root)

            else:

                root.left = self.\_rotate\_left(root.left)

                return self.\_rotate\_right(root)

        if balance\_factor < -1:

            if data > root.right.data:

                return self.\_rotate\_left(root)

            else:

                root.right = self.\_rotate\_right(root.right)

                return self.\_rotate\_left(root)

        return root

    def \_get\_height(self, root):

        if not root:

            return 0

        return max(self.\_get\_height(root.left), self.\_get\_height(root.right)) + 1

    def \_get\_balance\_factor(self, root):

        if not root:

            return 0

        return self.\_get\_height(root.left) - self.\_get\_height(root.right)

    def \_rotate\_left(self, z):

        y = z.right

        if not y:

            return z

        T2 = y.left

        y.left = z

        z.right = T2

        return y

    def \_rotate\_right(self, z):

        y = z.left

        if not y:

            return z

        T3 = y.right

        y.right = z

        z.left = T3

        return y

    def draw(self):

        self.\_draw\_helper(self.root, 0, 0, 20)

    def \_draw\_helper(self, root, x, y, spacing):

        if not root:

            return

        left\_x = x - spacing

        right\_x = x + spacing

        plt.text(x, y, str(root.data), style='italic', bbox={'facecolor': 'lightgreen', 'edgecolor': 'darkgreen', 'pad': 5})

        if root.left:

            plt.plot([x, left\_x], [y, y - 1], color='black')

            self.\_draw\_helper(root.left, left\_x, y - 1, spacing / 2)

        if root.right:

            plt.plot([x, right\_x], [y, y - 1], color='black')

            self.\_draw\_helper(root.right, right\_x, y - 1, spacing / 2)

# Solicitar al usuario ingresar los números enteros

avl\_tree = AVLTree()

numbers = []

while True:

    valor = input("Ingrese un número entero (o 'no' para terminar): ")

    if valor.lower() == 'no':

        break

    try:

        numero = int(valor)

        numbers.append(numero)

    except ValueError:

        print("Valor inválido. Ingrese un número entero.")

for number in numbers:

    avl\_tree.insert(number)

# Dibujar el árbol AVL final

plt.figure(figsize=(8, 6))

avl\_tree.draw()

plt.axis('off')

plt.show()



**Materia:** Programación Lógica Funcional

**Integrantes:** Vanessa Cruz Huitrón

**°°° ALGORITMO DE CLARKE Y WRIGHT °°°**

En temas anteriores se introdujo el problema VRP y se analizó con algunos tipos de restricciones que podían condicionar el problema. Uno de los algoritmos más conocidos para resolver el problema VRP es el algoritmo **Clark y Wright**, también conocido como **Algoritmo de los Ahorros (**Savings Algorithm**)**. Este algoritmo usa heurística la cual se analizará a continuación, para ir construyendo sistemáticamente las rutas de cada vehículo.

Evidentemente, hay que adaptar el algoritmo a las restricciones impuestas por el problema que se quiere resolver: ¿Se tiene un número limitado de vehículos?, ¿Hay un límite de kilómetros para los vehículos?, ¿Hay que cumplir horarios de reparto?, las restricciones son innumerables. Para atacar el problema se puede partir de los siguientes supuestos

* Se dispone de un único almacén donde parten y regresan todos los vehículos tras realizar su reparto.
* Por simplicidad, se va a suponer que no se tiene limitación en el número de vehículos disponibles.
* Tampoco hay límite de kilómetros para los vehículos.
* Todos los vehículos tienen un límite de carga igual, que no puede sobrepasarse.
* Cada cliente hace un pedido con peso concreto y puede ser diferente al resto.

Para mantener la generalidad, no se introducirá más restricciones espaciales y temporales.

En cualquier caso, introducir otro tipo de algoritmo. Hay que hacer notar que este algoritmo puede adaptarse perfectamente bien a la optimización de otros problemas similares, como el reparto de prensa, la recolección de basura, o planificación de transporte público a la hora de crear líneas de autobús.

La búsqueda utiliza el algoritmo de los ahorros, que se basa en el hecho de que, dado los clientes servidos por dos vehículos diferentes, si se forza a que solo un vehículo sino a los dos, se obtendrá un ahorro tanto en diferencia como en vehículos.

Si se considera N clientes que son servidos por N vehículos, es decir cada cliente servido por un solo vehículo. En este caso la distancia total recorrida por todos los vehículos es igual a la suma de las distancias desde el almacén hasta cada cliente y multiplicada por 2, ya que el vehículo debe ir y volver al almacén (suponiendo que el camino de ida tenga la misma distancia, que el de vuelta). Matemáticamente se puede expresar como:

Donde la función d (almacén, cliente i) representa la distancia entre el almacén y el i-esimo cliente. Aunque es una solucion valida, evidentemente est5a solucion es poco práctica. No es confiable usar un vehículo para cada cliente. Si se hace que uno de los vehículos, sirva a dos clientes (se le llamara cliente i y cliente j), se habrá ahorrado de entrada un vehículo, pero también se habrá ahorrado en distancia recorrida.

A ese ahorro se le llamará **s** y podrá ser calculado con la siguiente fórmula) siendo **A** el almacén).

**s(i , j) = 2d(A , i) + 2d(A , j) – [d(A , i) +d(A , j)]**

simplificando queda de la siguiente manera:

**s(i , j) = d(A , i) + d(A , j) – d(l , j)**

Ahora se puede ver de forma más intuitiva en el siguiente grafo:

Es decir, que **s** representa el ahorro de combinar a los clientes i y j en una sola ruta. El algoritmo de los ahorros trata de ir combinando en una misma ruta a los clientes, de forma que se produzca el mayor ahorro posible. Siempre y cuando no se viole ninguna restricción (por ejemplo, que el vehículo sobrepase su máximo de carga).

Supóngase que se dan las siguientes distancias:

* d(A, i) =10 km
* d(A, j) = 15 km
* d(i, j) = 12 km

El ahorro obtenido de unir a ambos clientes en una sola ruta es:

**s(i , j) = d(A , i) + d(A , j) – d(l , j)= 10 + 15 – 12 = 13**

Estos son los pasos que sigue el algoritmo de los ahorros para generar las rutas:

**Paso 1:** Calcular el ahorro s(i ,j) = d(A, i) + d(A, j) – d(i, j) para cada par de clientes.

**Paso 2:** Ordenar los ahorros de mayor a menor en la lista que se llamara, lista de ahorro.

**Paso 3:** Recordar la lista de ahorros. Por cada ahorro en la lista, si no se viola ninguna restricción.

* Si ni **i** ni **j** pertenecen a una ruta, crear una ruta con **i** y con **j.**
* Si **i** o **j** estan en ruta (pero no los dos) y además es el primero o el ultimo cliente de la ruta (son exteriores), se añade el cliente a la ruta de forma que **i** y **j** queden juntos.

**Paso 4:** Si quedan vehículos sin asignar a una ruta, se crea una o varias (según las restricciones) que los incluya.

En el siguiente programa implementa el algoritmo de los ahorros sobre la red de carreteras del ejemplo que se ha venido usado durante los temas anteriores. En un diccionario llamado coord. Se introduce las coordenadas de las ciudades. Las distancias entre las ciudades se calculan con la función distancia (). Por simplificar, se usará las distancias planas en línea recta que hay entre dos puntos del plano. Se tendrá que desempolvar a Pitágoras y se usara la formula.

Cada ciudad tiene una demanda **n** que se almacena en el diccionario llamado pedidos. Se supone que la carga máxima de un vehículo es de 40 cajas y se sitúa en el almacén en las coordenadas (19.984146,

-99.519127), en Jiloyork (Pueblo Mágico).

#VPR

import math

from operator import itemgetter

def distancia(coord1, coord2):

    lat1 = coord1[0]

    lon1 = coord1[1]

    lat2 = coord2[0]

    lon2 = coord2[1]

    return math.sqrt((lat1 - lat2)\*\*2 + (lon1 - lon2)\*\*2)

def en\_ruta(rutas, c):

    ruta = None

    for r in rutas:

        if c in r:

            ruta = r

    return ruta

def peso\_ruta(ruta):

    total = 0

    for c in ruta:

        total = total + pedidos[c]

    return total

def vrp\_voraz():

    #Calcular los ahorros

    s={}

    for c1 in coord:

        for c2 in coord:

            if c1 != c2:

                if not (c2,c1) in s:

                    d\_c1\_c2 = distancia(coord[c1], coord[c2])

                    d\_c1\_almacen = distancia (coord[c1], almacen)

                    d\_c2\_almacen = distancia (coord[c2], almacen)

                    s[c1,c2] = d\_c1\_almacen + d\_c2\_almacen - d\_c1\_c2

    #Ordenar Ahorros

    s = sorted(s.items(), key = itemgetter(1), reverse = True)

    #Construir rutas

    rutas = []

    for k,v in s:

        rc1 = en\_ruta(rutas, k[0])

        rc2 = en\_ruta(rutas, k[1])

        if rc1 == None and rc2 ==None:

            #No estan en nunguna ruta. Crear la ruta.

            if peso\_ruta([k[0], k[1]]) <= max\_carga:

                rutas.append([k[0], k[1]])

        elif rc1 != None and rc2 == None:

            #Ciudad 1 ya esta en ruta. Agregar la ciudad 2

            if rc1[0] == k[0]:

                if peso\_ruta(rc1) + peso\_ruta([k[1]]) <= max\_carga:

                    rutas[rutas.index(rc1)].insert(0, k[1])

            elif rc1[len(rc1)-1]== k[0]:

                if peso\_ruta(rc1) + peso\_ruta([k[1]]) <= max\_carga:

                    rutas[rutas.index(rc1)].append(k[1])

        elif rc1 == None and rc2 != None:

            #Ciudad 2 ya esta en ruta. Agregar la ciudad 1

            if rc2[0] == k[1]:

                if peso\_ruta(rc2) + peso\_ruta([k[0]]) <= max\_carga:

                    rutas [rutas.index(rc2)].insert(0,k[0])

            elif rc2[len(rc2) - 1] == k[1]:

                if peso\_ruta(rc2) + peso\_ruta([k[0]]) <=max\_carga:

                    rutas[rutas.index(rc2)].append(k[0])

        elif rc1 != None and rc2 != None and rc1 != rc2:

            #Ciudad 1 y 2 ya estan en una ruta.

            if rc1[0] == k[0] and rc2[len(rc2) -1] == k[1]:

                if peso\_ruta(rc1) + peso\_ruta(rc2) <= max\_carga:

                    rutas[rutas.index(rc2)].extend(rc1)

                    rutas.remove(rc1)

            elif rc1[len(rc1) -1] == k[0] and rc2 [0] == k[1]:

                if peso\_ruta(rc1) + peso\_ruta(rc2) <= max\_carga:

                    rutas[rutas.index(rc1)].extend(rc2)

                    rutas.remove(rc2)

    return rutas

if \_\_name\_\_ == "\_\_main\_\_":

    coord = {

        'JiloYork': (19.984146, -99.519127),

        'Toluca': (19.286167856525594, -99.65473296644892),

        'Atlacomulco': (19.796802401380955, -99.87643301629244),

        'Guadalajara': (20.655773344775373, -103.35773871581326),

        'Monterrey': (25.675859554333684, -100.31405053526082),

        'Cancún': (21.158135651777727, -86.85092947858692),

        'Morelia': (19.720961251258654, -101.15929186858635),

        'Aguascalientes': (21.88473831747085, -102.29198705069501),

        'Queretaro': (20.57005870003398, -100.45222862892079),

        'CDMX': (19.429550164848152, -99.13000959477478)

    }

    pedidos = {

        'JiloYork':10,

        'Toluca':15,

        'Atlacomulco': 0,

        'Guadalajara': 0,

        'Monterrey': 40,

        'Cancún': 50,

        'Morelia': 25,

        'Aguascalientes': 45,

        'CDMX':60,

        'Queretaro': 100

    }

    almacen = (40.23, -3.40)

    max\_carga = 40

    rutas = vrp\_voraz()

    for ruta in rutas:

        print (ruta)

**Hill Climbing**

También conocido como algoritmo de la Escalada de la Colina es un algoritmo de búsqueda local que, a partir de un estado inicial, trata de ir mejorando el resultado analizando estados vecinos. Siempre que se encuentre un estado vecino mejor que el actual (según la función de evaluación que se use), este estado sustituye al actual a partir de ese momento y se vuelve a repetir la operación con la vecindad del nuevo estado. El proceso termina cuando no es posible realizar mas mejoras. También se suele poner un límite de interacciones del algoritmo para evitar caer en demoras en tiempos de proceso.

El nombre del algoritmo proviene del hecho de que, si se imagina el espacio de estados como una función similar al de *ahorro producido al eliminar un vehículo,* el algoritmo se moverá por el espacio de estados hacia la pendiente creciente si lo que se quiere es encontrar el mínimo de la función, como si se estuviera subiendo una colina.

Durante la ejecución del algoritmo, no se pueden realizar más mejoras, esto es porque se alcanza el óptimo global, que es lo que se desea, o que se ha alcanzado un máximo local.

Como no es posible conocer el error optimo conocido con respecto al optimo global, ya que no se conoce su valor, no es posible saber en ningún momento si se ha caído en el máximo local.

Al encontrar un máximo local o global depende únicamente del estado inicial. Otro problema es que no es posible dar el limite superior del tiempo de ejecución del algoritmo, aunque si se puede limitar el numero de iteraciones, en cuyo caso se obtiene una solución que podría estar relativamente cercana a la óptima o no.

A pesar de los problemas que se puedan presentar, el algoritmo Hill Clinmbing, tiene la ventaja de ser muy simple de implementar. Solo se necesita una función de evaluación y un operador que permita generar los estados vecinos.

El seudocódigo del algoritmo hill clinmbing es:

*Función Hill\_climbing(estado\_actual):*

*Mejora = True*

*Minetras mejora == True:*

*Mejora = False*

*Genrar vecinos de estado\_actual*

*Por cada estado vecino:*

*Si nuevo\_estado mejora a estado\_actual:*

*Estado\_actual=nuevo\_estado*

*Mejora = True*

*Salir del estado actual*

Como ejemplo se tratara de resolver el problema TSP sobre el mapa de ciudades que han venido usando asta ahora. Como representación interna de la ruta se usara una lista con las cuidades vicitadas y con su orden de visita correspondiente. Por ejemplo, una ruta de 3 ciudades podría ser:

JilloYork -Qro-SLP

Se ha dicho que se, necesita una función de evaliacion. Parece que la distancia total de la ruta (la suma de todas las distancias entre ciudades consecutivas) es una función de valuaciion mas adecuada. Como operador para obtener los vecinos de la ruta se tendrá que definir operador llamada **intercambio , que intercambia dos ciudades de la ruta. Por ejemplo, las siguientes serian vecinas de l ruta queese han puesto de ejemplo:**

**QRO – JiloYork –SLP**

**SLP – QRO -JiloYork**

**JiloTork**

import math

import random

def distancia(coord1, coord2):

    lat1 = coord1[0]

    lon1 = coord1[1]

    lat2 = coord2[0]

    lon2 = coord2[1]

    return math.sqrt((lat1 - lat2) \*\* 2 + (lon1 - lon2) \*\* 2)

# Calcular la distancia total de una ruta

def evalua\_ruta(ruta):

    total = 0

    for i in range(len(ruta) - 1):

        ciudad1 = ruta[i]

        ciudad2 = ruta[i + 1]

        total += distancia(coord[ciudad1], coord[ciudad2])

    ciudad1 = ruta[-1]

    ciudad2 = ruta[0]

    total += distancia(coord[ciudad1], coord[ciudad2])

    return total

def hill\_climbing():

    # Crear la ruta inicial aleatoria

    ruta = list(coord.keys())

    random.shuffle(ruta)

    mejora = True

    while mejora:

        mejora = False

        distan\_actual = evalua\_ruta(ruta)

        # Evaluar vecinos

        for i in range(len(ruta)):

            if mejora:

                break

            for j in range(len(ruta)):

                if i != j:

                    ruta\_tmp = ruta[:]

                    ciudad\_tmp = ruta\_tmp[i]

                    ruta\_tmp[i] = ruta\_tmp[j]

                    ruta\_tmp[j] = ciudad\_tmp

                    dist = evalua\_ruta(ruta\_tmp)

                    if dist < distan\_actual:

                        # Se ha encontrado un vecino que mejora la solución

                        mejora = True

                        ruta = ruta\_tmp[:]

                        break

    return ruta

if \_\_name\_\_ == "\_\_main\_\_":

    coord = {

        'JiloYork': (19.984146, -99.519127),

        'Toluca': (19.286167856525594, -99.65473296644892),

        'Atlacomulco': (19.796802401380955, -99.87643301629244),

        'Guadalajara': (20.655773344775373, -103.35773871581326),

        'Monterrey': (25.675859554333684, -100.31405053526082),

        'Cancún': (21.158135651777727, -86.85092947858692),

        'Morelia': (19.720961251258654, -101.15929186858635),

        'Aguascalientes': (21.88473831747085, -102.29198705069501),

        'Queretaro': (20.57005870003398, -100.45222862892079),

        'CDMX': (19.429550164848152, -99.13000959477478)

    }

    ruta = hill\_climbing()

    print(ruta)

    print("Distancia Total: " + str(evalua\_ruta(ruta)))

Existe una variacion del algoritmo de hill climbing que trata de combinar los problemas relativos a los óptimos locales. Se trata de **Hill Climbing interactivo. La idea de ejecutar varias veces el algoritmo, pero partiendo de un estado inicial diferente cada vez. El siguiente código realiza 10 ejecuciones (Variable max\_iteracciones) y al final de cada ejecución se compara el resultado con la variable mejor\_ruta, que almacena la mejor ruta encontrada hasta el momento. Si la ruta encontrada en una de las iteraciones es mejor que la almacenada en mejor\_rutas, se sustituye por la nueva mejor.**

import math

import random

def distancia(coord1, coord2):

    lat1 = coord1[0]

    lon1 = coord1[1]

    lat2 = coord2[0]

    lon2 = coord2[1]

    return math.sqrt((lat1 - lat2) \*\* 2 + (lon1 - lon2) \*\* 2)

# Calcular la distancia correcta por una ruta

def evalua\_ruta(ruta, coord):

    total = 0

    for i in range(0, len(ruta) - 1):

        ciudad1 = ruta[i]

        ciudad2 = ruta[i + 1]

        total += distancia(coord[ciudad1], coord[ciudad2])

    ciudad1 = ruta[-1]

    ciudad2 = ruta[0]

    total += distancia(coord[ciudad1], coord[ciudad2])

    return total

def i\_hill\_climbing(coord):

    # Crear ruta inicial aleatoria

    ruta = list(coord.keys())  # Inicializa la ruta con todas las ciudades

    mejor\_ruta = ruta[:]

    max\_iteraciones = 10

    while max\_iteraciones > 0:

        mejora = False

        # Genera nueva ruta aleatoria

        random.shuffle(ruta)

        for i in range(len(ruta)):

            for j in range(i + 1, len(ruta)):

                ruta\_tmp = ruta[:]

                ruta\_tmp[i], ruta\_tmp[j] = ruta\_tmp[j], ruta\_tmp[i]

                dist = evalua\_ruta(ruta\_tmp, coord)

                if dist < evalua\_ruta(ruta, coord):

                    # Se encuentra un vecino que mejora el resultado

                    mejora = True

                    ruta = ruta\_tmp[:]

        max\_iteraciones -= 1

        if evalua\_ruta(ruta, coord) < evalua\_ruta(mejor\_ruta, coord):

            mejor\_ruta = ruta[:]

    return mejor\_ruta

if \_\_name\_\_ == "\_\_main\_\_":

    coord = {

        'JiloYork': (19.984146, -99.519127),

        'Toluca': (19.286167856525594, -99.65473296644892),

        'Atlacomulco': (19.796802401380955, -99.87643301629244),

        'Guadalajara': (20.655773344775373, -103.35773871581326),

        'Monterrey': (25.675859554333684, -100.31405053526082),

        'Cancún': (21.158135651777727, -86.85092947858692),

        'Morelia': (19.720961251258654, -101.15929186858635),

        'Aguascalientes': (21.88473831747085, -102.29198705069501),

        'Queretaro': (20.57005870003398, -100.45222862892079),

        'CDMX': (19.429550164848152, -99.13000959477478)

    }

    mejor\_ruta = i\_hill\_climbing(coord)

    print(mejor\_ruta)

    print("Distancia total:", evalua\_ruta(mejor\_ruta, coord))

Al ejecutar varias veces esta nueva versión, se puede observar que el resultado es mucho mas coherente cada vez que se compila (con suerte se obtendrá el mismo). Aun así asegura que se haya alcanzado el máximo global

**Simulated Annealing**

Como se ha visto en la sección anterior, el mayor peligro con el que se enfrenta la búsqueda local es caer en un máximo local. Las tres técnicas que se vana presentar en lo que queda del curso, se van a enfocar a intentar escapar de los máximos locales. Son **metaheurísticos.**

(Métodos heurísticos genéricos para problemas que no disponga de un algoritmo que os resuelva parametrizables) que permiten atacar un espectro de problemas diferentes. Las tres técnicas que se van a analizar son el **templado, Simulado o Simulated Annealing, la Busqueda tabú y los algoritmos genéticos evolutivos.**

La técnica del templado simulado, tiene su origen en el proceso físico del templado del acero, el cristal o la cerámica. La técnica del templando consiste en elevar mucho la temperatura del material y luego enfriarlo lentamente para alterar sus propiedades físicas y hacerlos mas resientes.

El templado simulado se parece mucho a la técnica de Hill climbing pero añade algunas variaciones. La primera diferencia es cada iteración, no siempre elegirá a un vecino que mejora la solución actual, si no que puede selección peor. La aceptación o no de una solución depende de una función probabilista. De esta forma no siempre se tomara el camino de mayor pendiente como se hacía en Hill climbig, sino que se añade un elemento de variación que permitara escapar de los máximos locales.

Dut¿rante el proceso, en lugar de evaluar todos los vecinos, del estado actual, se evalua solo algunos. Si el nuevo estado mejra el actual , se hará el cambio tal y como se hacia en Hill climbing, pero si no, se usara la función de probabilidad que decidirá si se hace o no el cambio.

¿Qué función de probabilidad se debe usara?

Básicamente la probabilidad de hacer el cambio dependerá de dos factores, el primer factor es la inferencia relativa que hay entre el estado actual y el nuevo. Si la nueva solución es considerablemente pero que, la actual, tendrá menos probabilidad de realizar el cambio. El segundo factor se llamara temperatura y se denotara como T. la función de probabilidad que se usara es la siguiente:

Siendo Eactual el estado actual y Enuevo el nuevo estado vecino. La función eval() es la función de evaluación que se usa para ,medir la calidad de solución. Por ejemplo, la distancia de la ruta en caso de TSP. Si lo que se busca es el estado mínimo, se cambia el minuendo y el sustraendo del numerador del exponente.

El rol que realiza la operación (Eactual – Enuevo) es claro: la probabilidad de cambio depende de la medida en que una solución es peor que la otra (diferencia de costos), pero ¿que misión tiene T(temperatura)? En la siguiente tabla se observa que ocurre al ir incrementando el valor de T.

|  |  |
| --- | --- |
| T | P = e |
| 1 | 0.00004 |
| 5 | 0.13533 |
| 10 | 0.36787 |

Se observa que, a mayor valor de T, mayor es el valor de la función de probabilidad. Esto quiere decir que ha un valor alto de la temperatura favorecerá que se produzca la selección de un estado peor que el actual.

La técnica se llama templado simulado porque consiste en comenzar con un valor alto de temperatura T e ir disminuyendo poco a poco en cada interacción, de forma que los primeros momentos, sea fácil que seleccione las soluciones peores, y con el paso del tiempo, según se enfrié T, ir estabilizándose en la mejor solución encontrada, ya que la probabilidad de seleccionar un vecino peor decrece.

El pseudocodigo del algoritmo es el siguiente:

Función simulated\_annealing(Estado\_actual):

Inicializar T

Inicializar T\_Min

Inicializar v\_enfriamiento.

Mientras T < T\_Min

Para i = 0 hasta v\_enfriamiento:

Estado\_nuevo = elegir un estado vecino

Si eval(estado\_actual) < eval(estado\_nuevo):

Estado\_actual = estado\_nuevo

Si no:

Si random(0,1) <

Estado\_actual = estado\_nuevo

Enfriar(T)

Salir con el estado\_actual

Se puede notar en las 3 variables que se inicializa desde el principio de la función la variable T(temperatura) la cual es conocida, ¿Pero qué valor inicial hay que darle? Un valor, muy alto haría que se tardara mucho en converger hacia la solución estable mientras que un valor bajo no permitiría, que la ejecución tuviera tiempo de alcanzar un estado estable aceptable.

La variable T\_Min indica un valor inferior que se alcance en un estado de enfriamiento y, por lo tanto, cuando la temperatura alcance el valor de T\_Min se abra terminado (es la condición de Stop). Es la práctica es habitual que el algoritmo termine al llegar un número máximo de interacciones en lugar de alcanzar el valor T\_Min.

Finalmente, la variable v\_enfriamiento (velocidad de enfriamiento) indica cuantos vecinos se evalúan en cada iteración antes de proceder a llamar a la función enfriar (). La velocidad de enfriamiento ha de ser lo suficiente grande para permitir que se alcance un estado estacionario para la temperatura de ese momento.

En general, la elección de los valores de estos tres parámetros dependerá de gran medida del problema en concreto y del tamaño de la vecindad de las soluciones.

Ya se ha comentado que la función () enfriar va disminuyendo el valor de la temperatura T en cada iteración, pero ¿con que velocidad hay que enfriar T?, existe diversos métodos, y la elección de uno u otro dependerá de nuevo, del problema sobre el que se este trabajando.

La forma mas intuitiva es enfriat T, disminuyéndola linealmente, es decir, ir restando una cantidad fija en cada iteración del bucle. Otras funciones que pueden ser utilizadas para el enfriamiento son:

|  |  |
| --- | --- |
| Descenso Exponencial | ) |
| Criterio de Boltzmann |  |
| Criterio de Cauchy |  |

import math

import random

def distancia(coord1, coord2):

    lat1, lon1 = coord1

    lat2, lon2 = coord2

    return math.sqrt((lat1 - lat2)\*\*2 + (lon1 - lon2)\*\*2)

def evalua\_ruta(ruta, coord):

    total = 0

    for i in range(len(ruta)):

        ciudad1 = ruta[i]

        ciudad2 = ruta[(i + 1) % len(ruta)]  # Utilizar el operador módulo para volver al inicio de la ruta

        total += distancia(coord[ciudad1], coord[ciudad2])

    return total

def simulated\_annealing(ruta, coord):

    T = 20

    T\_MIN = 0

    V\_enfriamiento = 100

    while T > T\_MIN:

        dist\_actual = evalua\_ruta(ruta, coord)

        for \_ in range(V\_enfriamiento):

            # Intercambio de dos ciudades aleatoriamente

            i = random.randint(0, len(ruta) - 1)

            j = random.randint(0, len(ruta) - 1)

            ruta\_tmp = ruta[:]

            ruta\_tmp[i], ruta\_tmp[j] = ruta\_tmp[j], ruta\_tmp[i]

            dist = evalua\_ruta(ruta\_tmp, coord)

            delta = dist - dist\_actual

            if dist < dist\_actual or random.random() < math.exp(-delta / T):

                ruta = ruta\_tmp

                dist\_actual = dist

        # Enfriar a T linealmente

        T -= 0.005

    return ruta

if \_\_name\_\_ == "\_\_main\_\_":

    coord = {

        'Jiloyork': (19.916012, -99.580580),

        'Toluca': (19.289165, -99.655697),

        'Atlacomulco': (19.799520, -99.873844),

        'Guadalajara': (20.677754472859146, -103.34625354877137),

        'Monterrey': (25.69161110159454, -100.321838480256),

        'QuintanaRoo': (21.163111924844458, -86.80231502121464),

        'Michohacan': (19.701400113725654, -101.20829680213464),

        'Aguascalientes': (21.87641043660486, -102.26438663286967),

        'CDMX': (19.432713075976878, -99.13318344772986),

        'QRO': (20.59719437542255, -100.38667040246602)

    }

    # Crear una ruta inicial aleatoria

    ruta = list(coord.keys())

    random.shuffle(ruta)

    ruta\_optima = simulated\_annealing(ruta, coord)

    print("Ruta óptima encontrada:")

    print(ruta\_optima)

    print("Distancia Total: " + str(evalua\_ruta(ruta\_optima, coord)))

BUSQUEDA TABÚ

La **búsqueda tabú,** al igual que la técnica del templado simulado, es un procedimiento metaheurístico que trata de diversificar la exploración en el espacio de estados para evitar caer en óptimos locales. Puede ocurrir que zonas menos prometedoras a priori conduzcan al optimo global. La búsqueda tabú utiliza una memoria adaptativa que va variando durante la ejecución del algoritmo y que le permite “recordar” los caminos visitados recientemente. A estos caminos se les llamara tabú cada iteración del algoritmo consigue explorar otros que, a priori, nunca se elegirían como prometedores. Es evidente que el algoritmo no seria útil si se marca el camino como tabú y ya nunca mas se vuelve a explorar por ahí. Se ha dicho que la memoria adaptativa almacena los caminos tomados mas recientemente, por lo que pasado el tiempo, aquellos caminos marcados como tabú deben dejar de serlo. Al numero de iteraciones que permanecerá una variable siendo tabú se llamara **tiempo de persistencia.**

Se comenzará analizando el problema SAT desde una nueva perspectiva. Supóngase que nuestro problema SAT tiene 5 variables booleanas tales que:

Se pude vntomar solamente los valores de 0 y 1. Una posible asignación de valores para estas variables podría ser

Se tiene que recordar que un vecino de V será cualquier otra asignación de valores para Vn tal que, solo se diferencien en el valor de una variable. Por ejemplo:

Es decir que el operador que se una para generar los estados vecinos en la búsqueda, será simplemente cambiar el valor de una de las variables. En el ejemplo anterior se han cambiado el valor de las dos ultimas variables, con lo que se ha obtenido una función llamada f2 evidentemente, la elección del mejor vecino vendrá dada por la función de evaluación que se este utilizando.

Dado este ecenario una búsqueda tabú evitaría, durante algunas iteraciones volver a cambiar el valor de las últimas dos variables, se necita una estructura de datos para almacenar que variable ha cambiado recientemente y además, durante cuanto tiempo ha de seguir siendo un camino tabú.

Supóngase que cuando se marca una variable como tabú estará prohibido cambiarla durante las siguientes tres iteraciones. Además, para almacenar esta información se va a utilizar un vector de enteros de longitud 5, que en un principio tendrá todos sus valores inicializados en 0. Si por ejemplo durante la búsqueda se selecciona un vecino en el cual se modifica el valor de la 5ta variable, se anotará el vector que durante las siguientes 3 iteraciones, dicho cambio no estará disponible y se hace colocando el valor 3 en la quinta posición del vector.

Si en las siguientes 3 iteraciones, el mejor vecino vuelve a ser el resultante de cambiar la quinta variable, al comprobar que dicha posición del vector es distinta a 0, se evitara hacer el cambio. Por su puesto, en cada iteración se deberá decrementar aquellas posiciones del vector con valores mayores a 0, de forma que en el ejemplo, al ocurrir 3 iteraciones el valor de la 5ta casilla vuelve a ser 0 y por lo tanto estaría de nuevo disponible. Continuamos con el ejemplo anterior del cual hemos partido

La memoria se inicializa en 0:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |

Supóngase que en la primera iteración, la función de evaluación indica que el mejor vecino es el que ha resultado de cambiar la quinta variable. En ese caso se selecciona como nuevo nodo actual y se marca como tabú la quinta variable durante las 3 próximas iteraciones.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 3 |

En la siguiente iteración, el algoritmo decide cambiar la tercera variable, así que se marca y se decrementan aquellas variables mayores que cero.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 3 | 0 | 2 |

En la siguiente interacción se decide el cambio de la segunda variable quedando el vector de la siguiente manera:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 3 | 2 | 0 | 1 |

En la siguiente interacción, la función de evaluación indica que el mejor cambio es la quinta variable; sin embargo, al ser mayor que cero y estar marcado como tabú, no es posible hacerlo. Como este cambio es tabú se toma el siguiente mejor, que en este caso podría ser el 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 2 | 1 | 0 | 0 |

Finalmente, el algoritmo selecciona de nuevo la quinta variable para el cambio. Ahora su valor es 0 por lo que si es posible seleccionarla.

Puede ocurrir que en algún estado vecino, incluso siendo tabú permita una solución excepcionalmente buena. En esta situación como se plantea, no conviene desperdiciar un estado tan prometedor, así, que bajo circunstancias especiales se permiten romper las reglas al criterio que permita decidir si se rompe o no las reglas, a esto se le llama **criterio de aspiración.** En la practica la búsqueda tabú recuerda siempre la mejor solución que ha encontrado desde el inicio de la búsqueda, y si el nuevo estado es mejor que el que este almacenado lo selecciona, aunque sea tabú.

El seudocódigo para el algoritmo de búsqueda tabú es:

Función busqueda\_tabu (estado\_actual):

Mejor\_estado = estado\_actual

Inicializar memoria\_tabú

Inicializar persistencia

Inicializar el número de iteraciones

Mientras numero\_iteraciones >=:

Numero\_iteraciones = numero\_iteaciones-1

Val\_actual = evaluar(estado\_actual)

Para cada vecino:

estado\_nuevo = elegir un estado vecino

Si estado\_nuevo no es tabú:

Si estado\_nuevo mejora a estado\_actual:

Estado\_actual = estado\_nuevo

Almacenar cambio en memoria tabú

Si estado\_actual es mejor que mejor\_estado:

Mejor\_estado = estado\_actual

Salir del bucle para

Si no:

Si estado\_nuevo menor que mejor\_estado:

Estado\_actual = estado\_nuevo

Almacenar cambio en memoria tabú

Mejor\_estado = estado\_actual

Salir del bucle para:

Decrementar presistencia de estado tabú

Salir con mejor ruta.

La idea básica de búsqueda tabú no es compleja en si misma, pero hay que adaptarla a cada problema correctamente. Si se quiere aplicar al TSP, la estructura de datos que se ha usado con el problema SAT ya no es valida

Se ha venido utilizando como operador para generar las rutas vecinas en el problema TSP el intercambio de ciudades. ¿Qué estructura de datos se puede usar para recordar que dos ciudades han sido intercambiadas recientemente? En este caso se puede usar una estructura de datos como la siguiente.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 5 | 6 | 7 |  |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
|  | 0 | 0 | 0 | 0 | 0 | 2 |
|  |  | 0 | 0 | 0 | 0 | 3 |
|  |  |  | 0 | 0 | 0 | 4 |
|  |  |  |  | 0 | 0 | 5 |
|  |  |  |  |  | 0 | 6 |

Los valores en la parte superior y en la lateral derecho representan ciudades, los valores de las casillas tienen el mismo significado, que el ejemplo del problema del SAT que se ha visto antes. Es decir que si se hace un intercambio, las ciudades dos y cinco y se quiere que sea tabú para las próximas tres interaciones, la estructura de datos quedaría asi:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 5 | 6 | 7 |  |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
|  | 0 | 0 | 3 | 0 | 0 | 2 |
|  |  | 0 | 0 | 0 | 0 | 3 |
|  |  |  | 0 | 0 | 0 | 4 |
|  |  |  |  | 0 | 0 | 5 |
|  |  |  |  |  | 0 | 6 |

Considerar que, por ejemplo, el intercambio entre las ciudades 2 y 5 es el mismo que el intercambio de las ciudades 5 y 2.

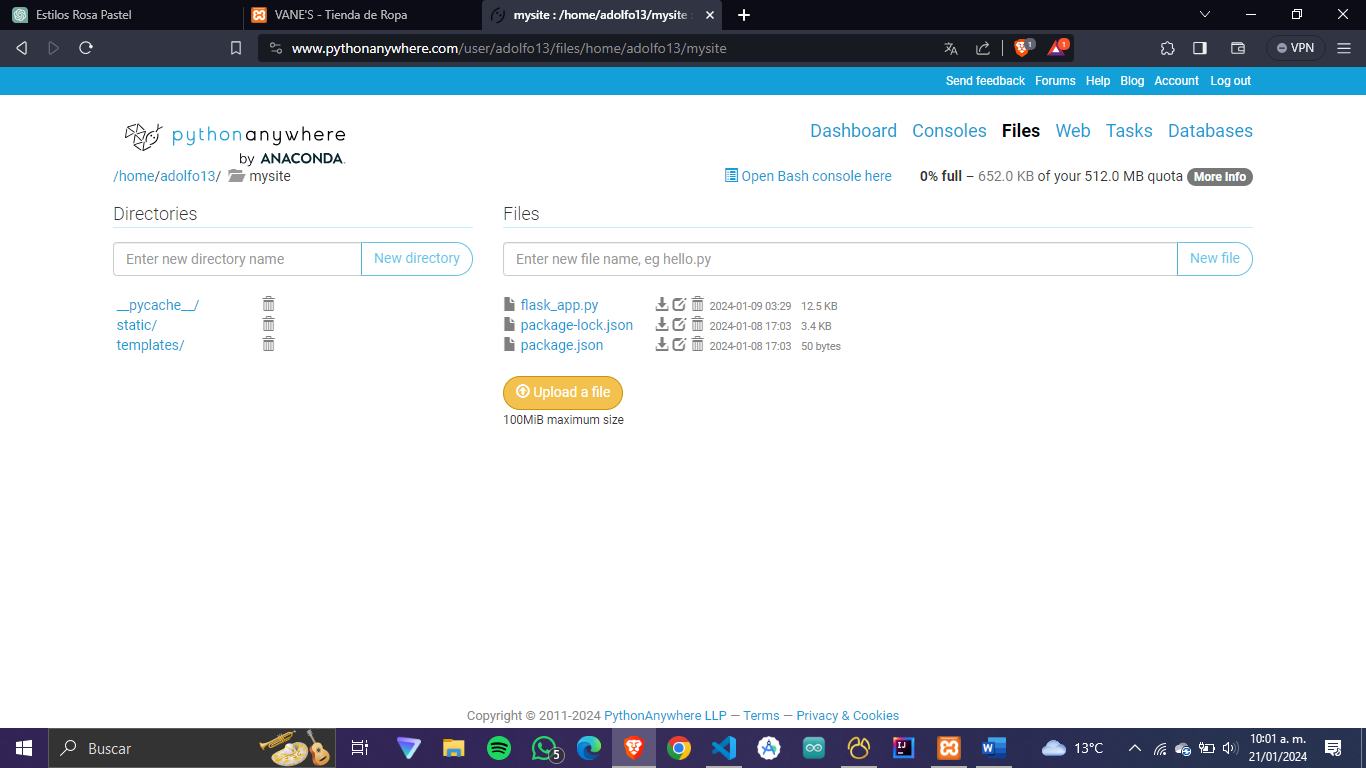
En la siguiente lista se va implementar el problema TSP, usando la búsqueda tabú. La estructura de datos presentada es conceptual, así que en vez de utilizar tal cual se ha escrito (se podría hacerlo sin problema), se deberá usar un diccionario de Python, en la que la clave será la pareja de ciudades intercambiadas y separadas por un guion bajo; y el valor, el número de iteraciones permanecerá siendo un movimiento tabú. Por ejemplo. Si se intercambia

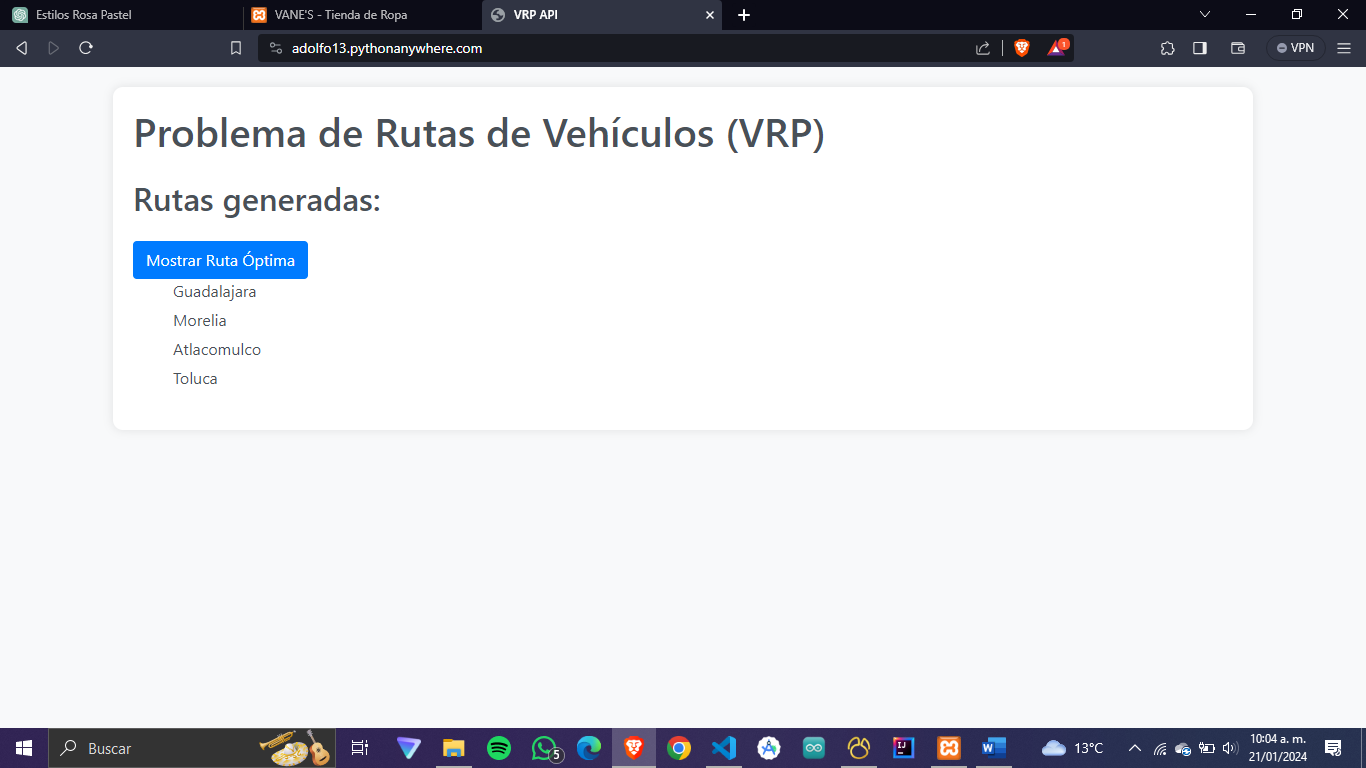
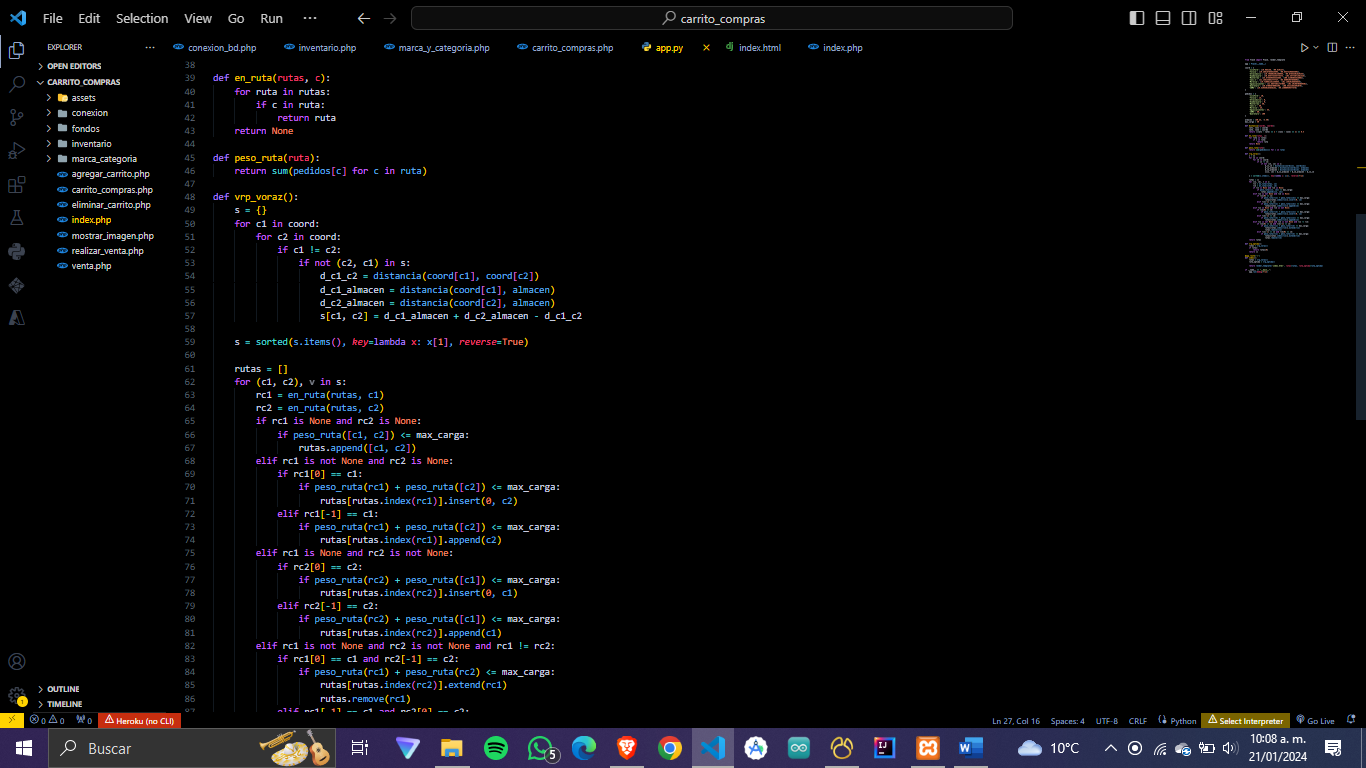
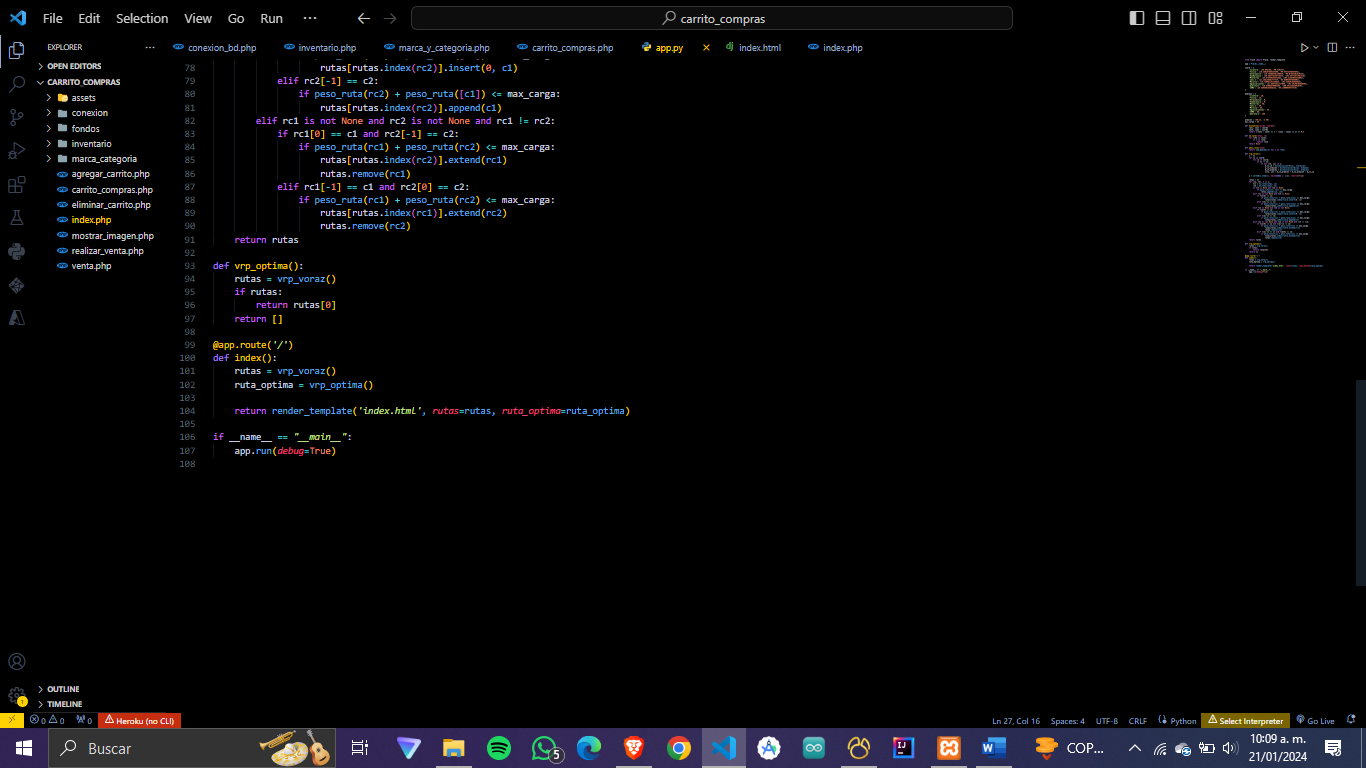
**PRACTICAS DEPLOYADAS EN FLASK Y PYTHON**

1. Primero debemos tener una cuenta donde se deployaran las APIS en este caso PythonAnywhere

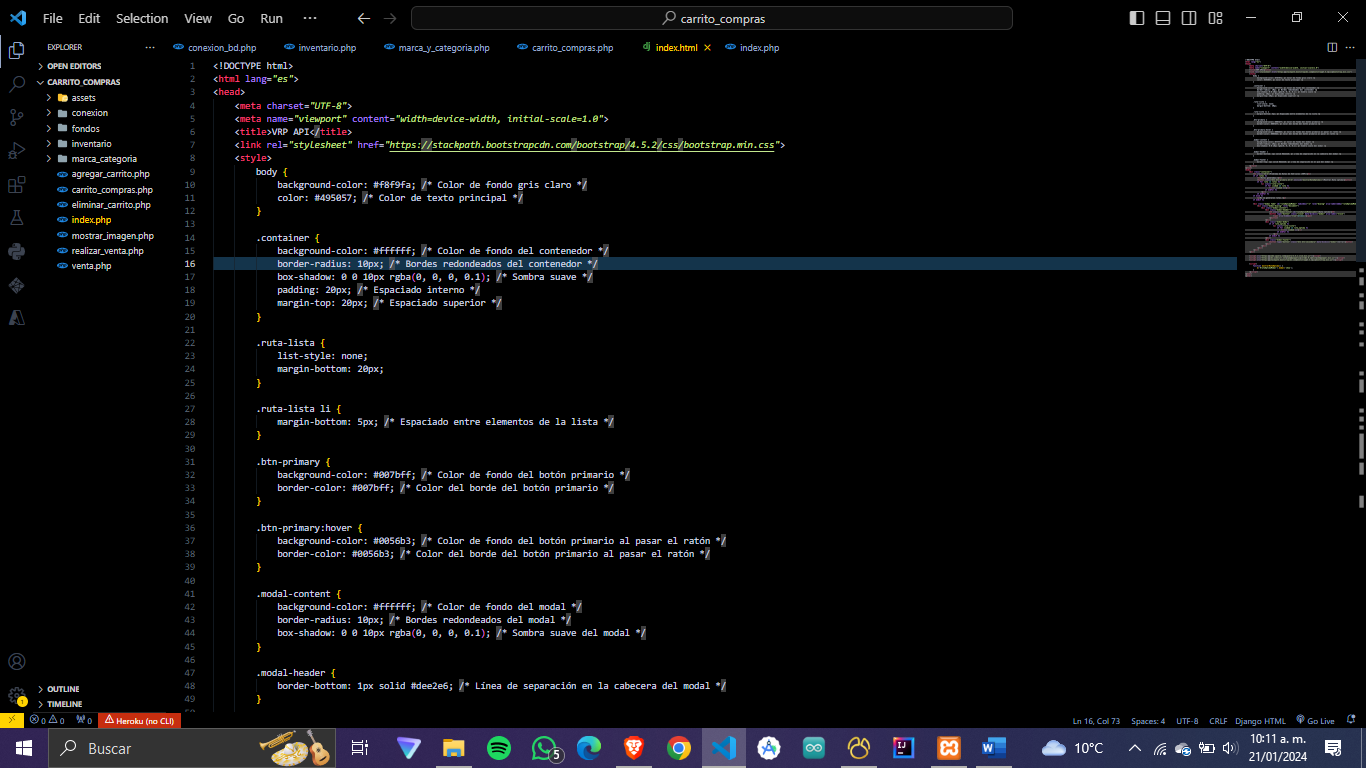
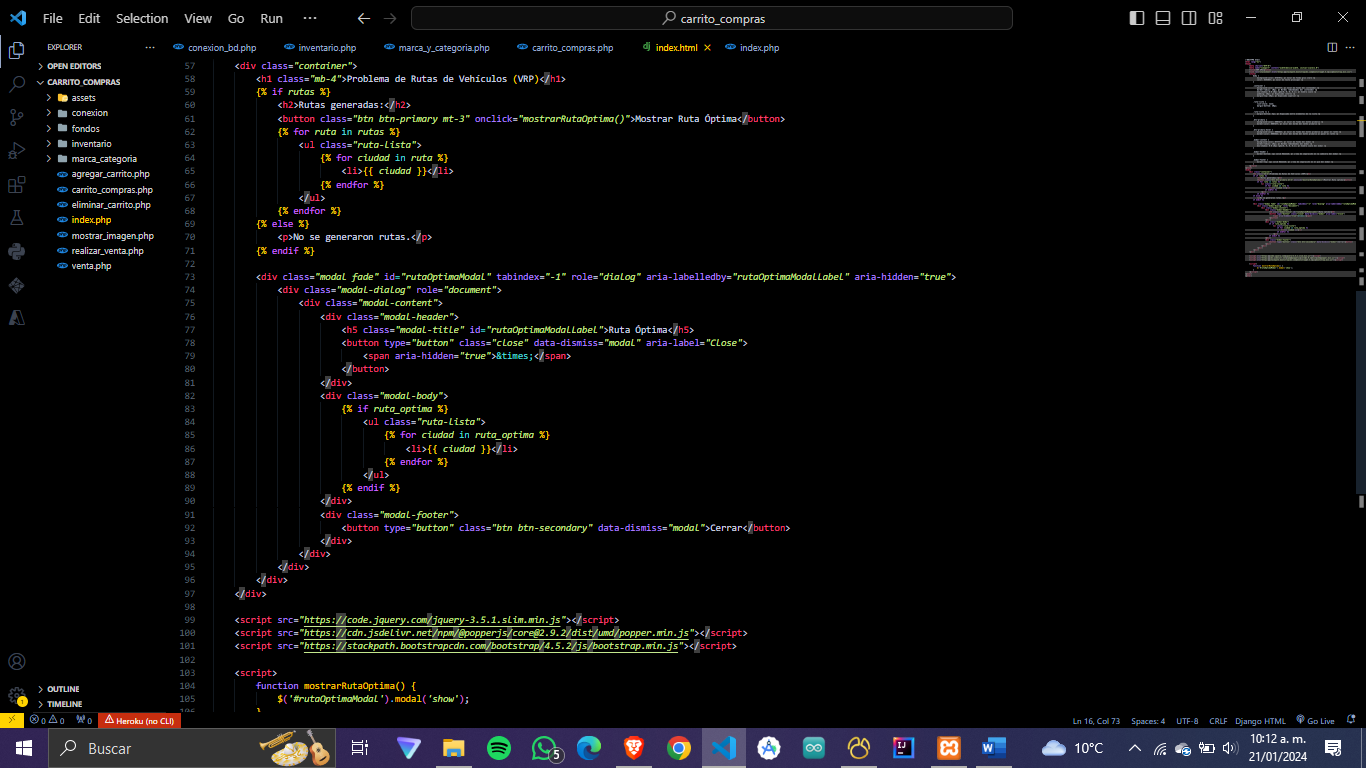


Deberías haber creado una web para poder deployar es necesario tener en cuenta que deveras tener una carpeta llamada Templates para poder tener el front de tu pagina a mostrar tu aplicación y el funcionamiento de aplicación es necesario referenciar tu aplicación como app.py para un mejor manejo de tu aplicación   
en esta parte para deplyar igual solo que cambio el nombre de tu archivo app a flask\_app.py

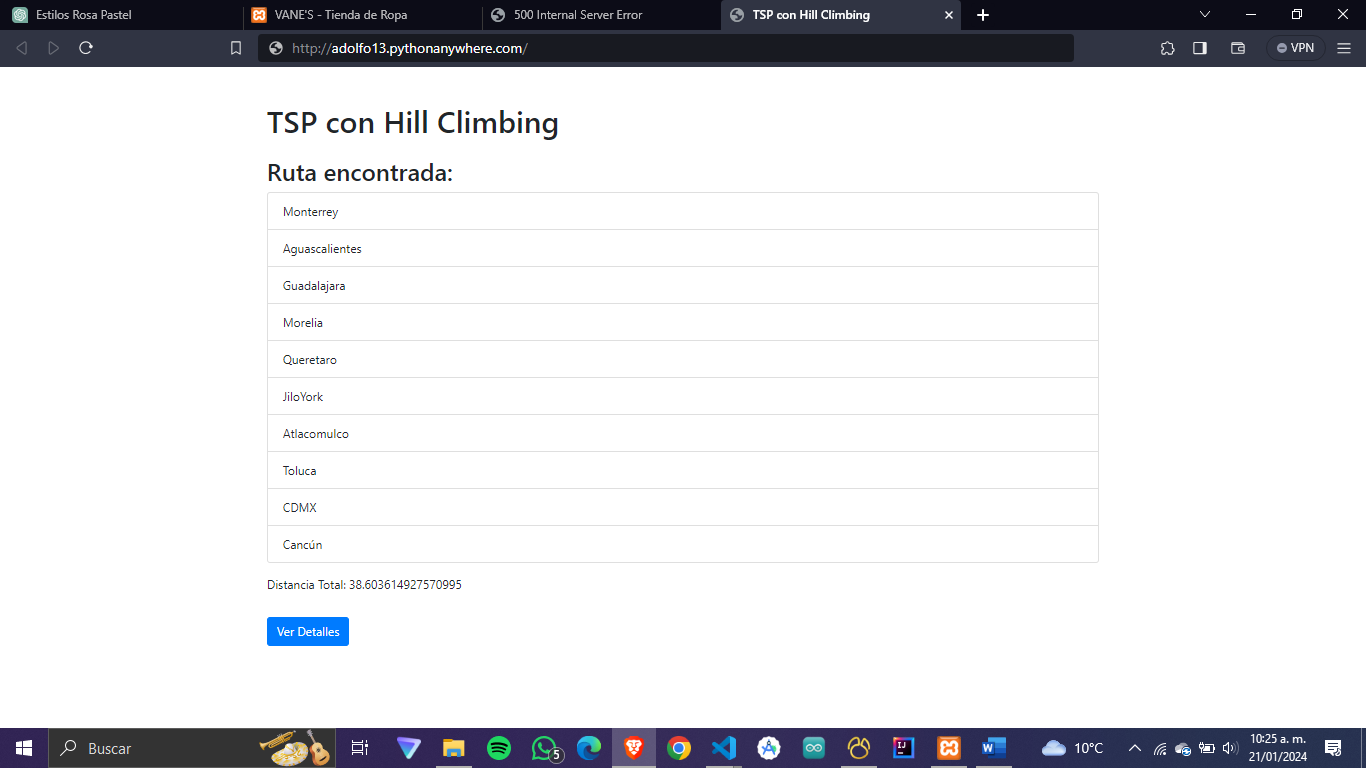


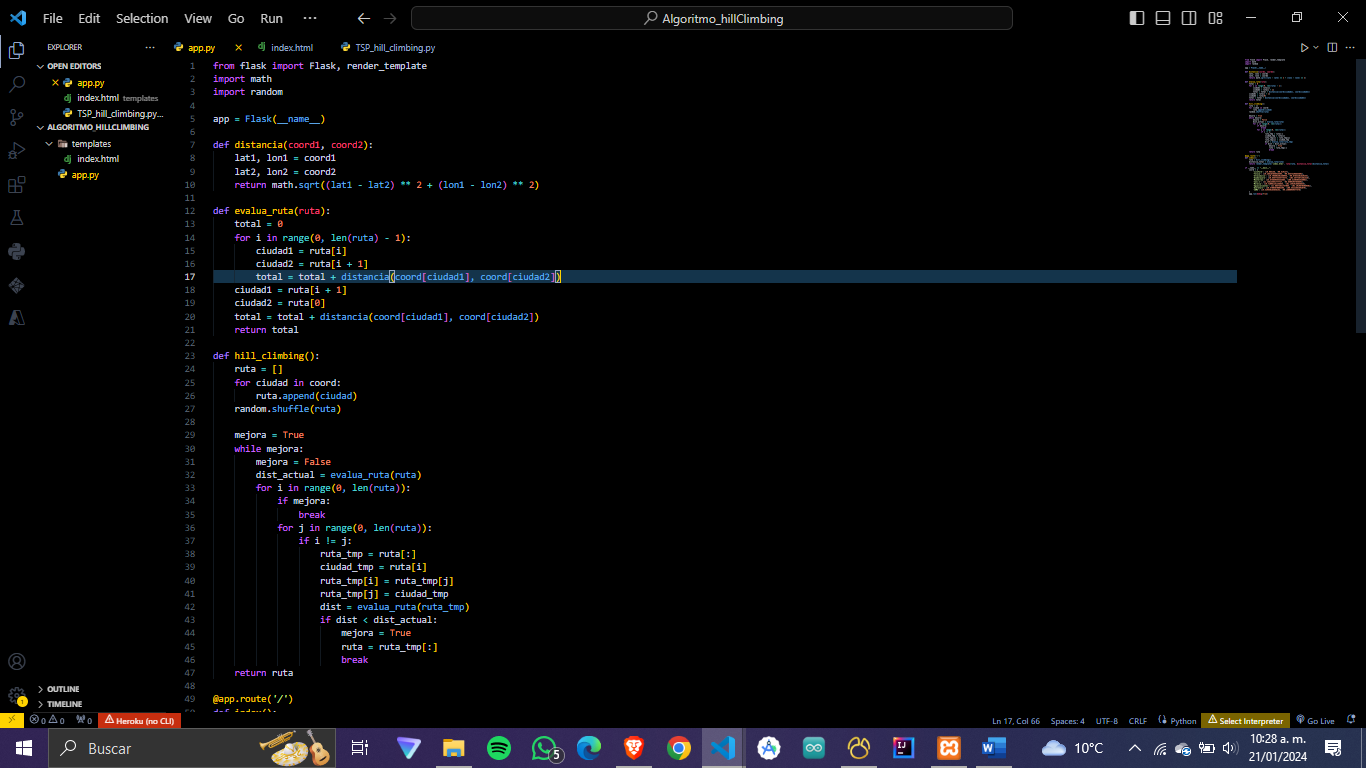
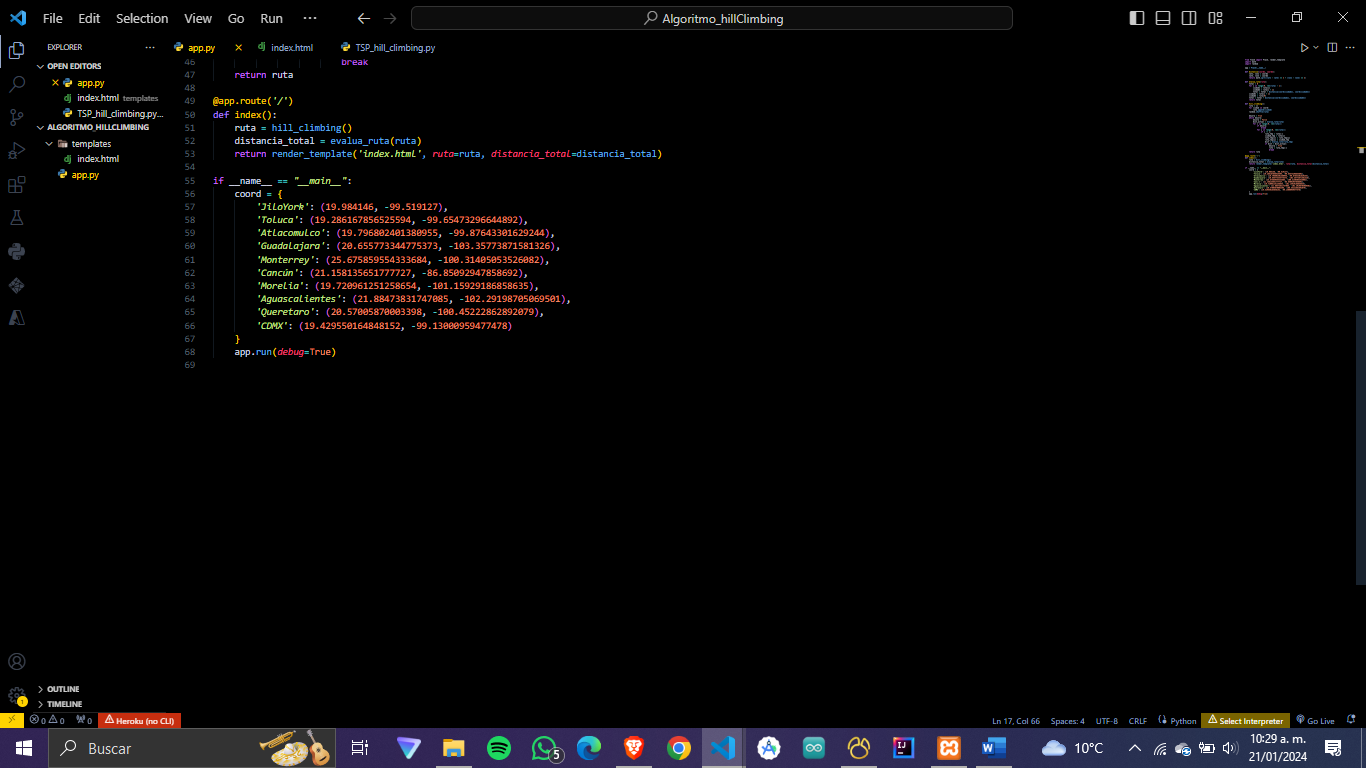
1.1 como primer trabajo tenemos el ALGORTMO C\_W  
donde nos muestra las rutas mas optimas de rutas de veichulos por VRP   
  
para logarar esto se tubo el código entrenado en Api en nuestra aplicación app.py que es este   
  

Esta parte de código es lo que hace que funcione la aplicación ahora bien para el front usamos un index en HTML done mandamos a llamar estas funciones

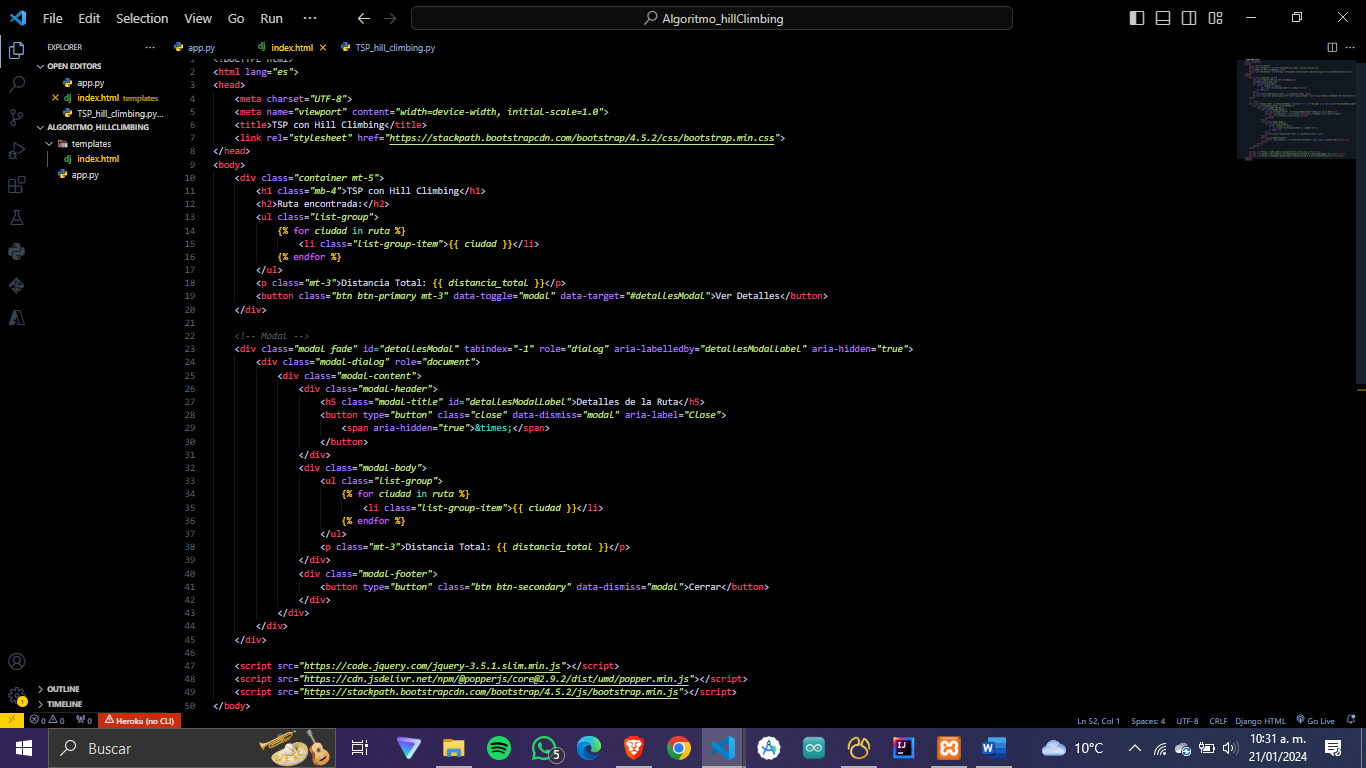
 

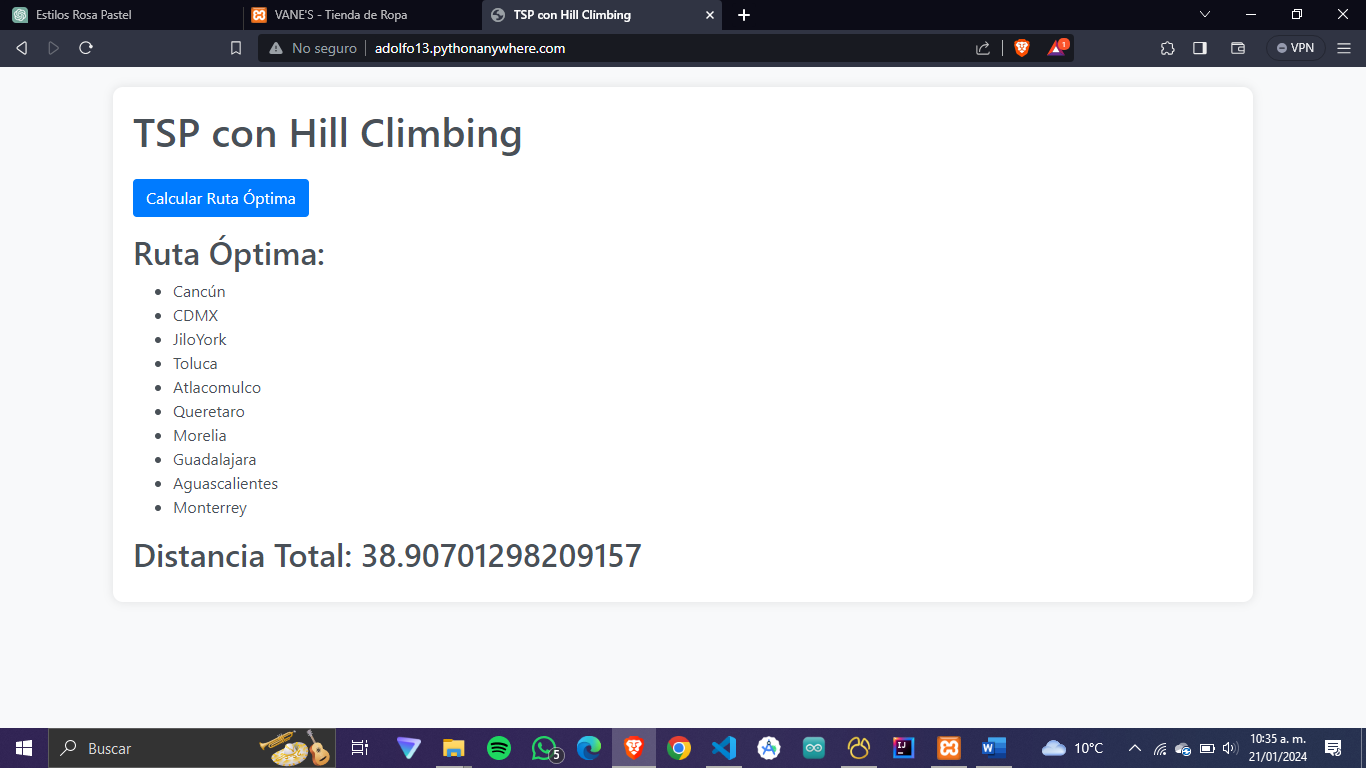
Estos códigos tal cual se subirán en la parte de PythonAnywhere para que funcionen como el ejemplo de arriba asi para que este deploya y cualquier persona con la URL puede visualizarlo desde cualquier parte u dispositivo

* 1. TSP con HILL CLINBING  
     

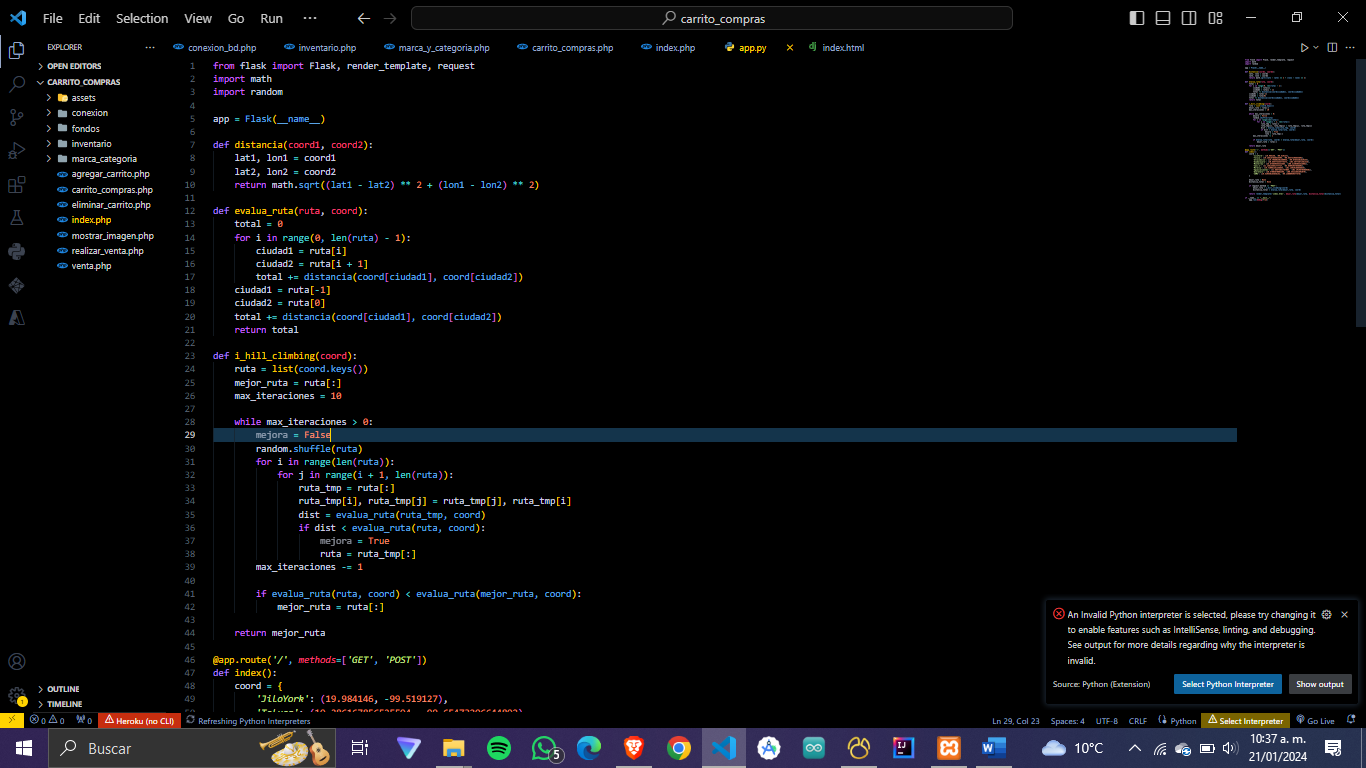
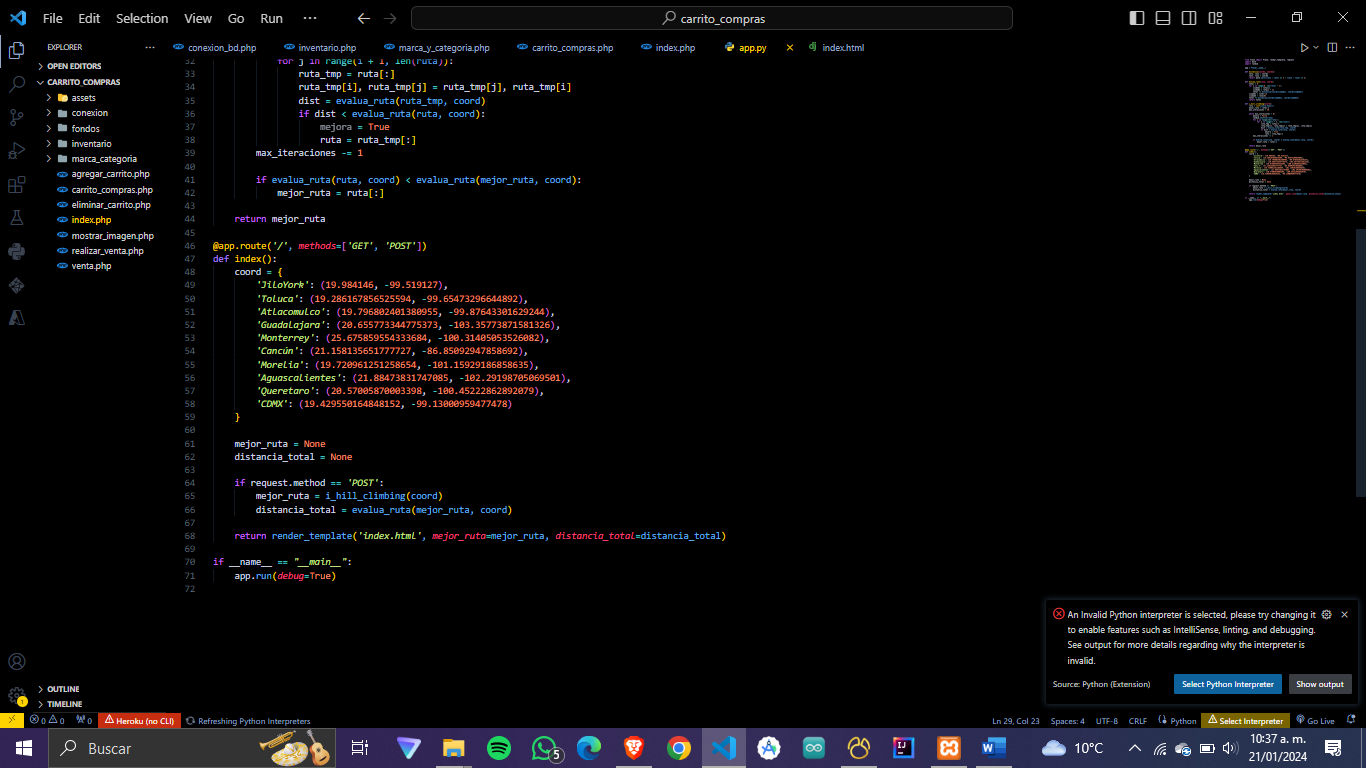
Esta aplicación lo que hace es buscar las rutas mas optimas para asi poder generar una distancia considerable mas corta y sin estar regresando de un punto a otro lo que hace es buscar las rutas mas cortas o con menos discrepancia para poder llegar a tiempo pongamos una ejemplo de los carritos de entrega deben de hacer entregas en un menor tiempo para cada cliente este es lo que hace y saca la mejor distancia que hay entre ellas  
 

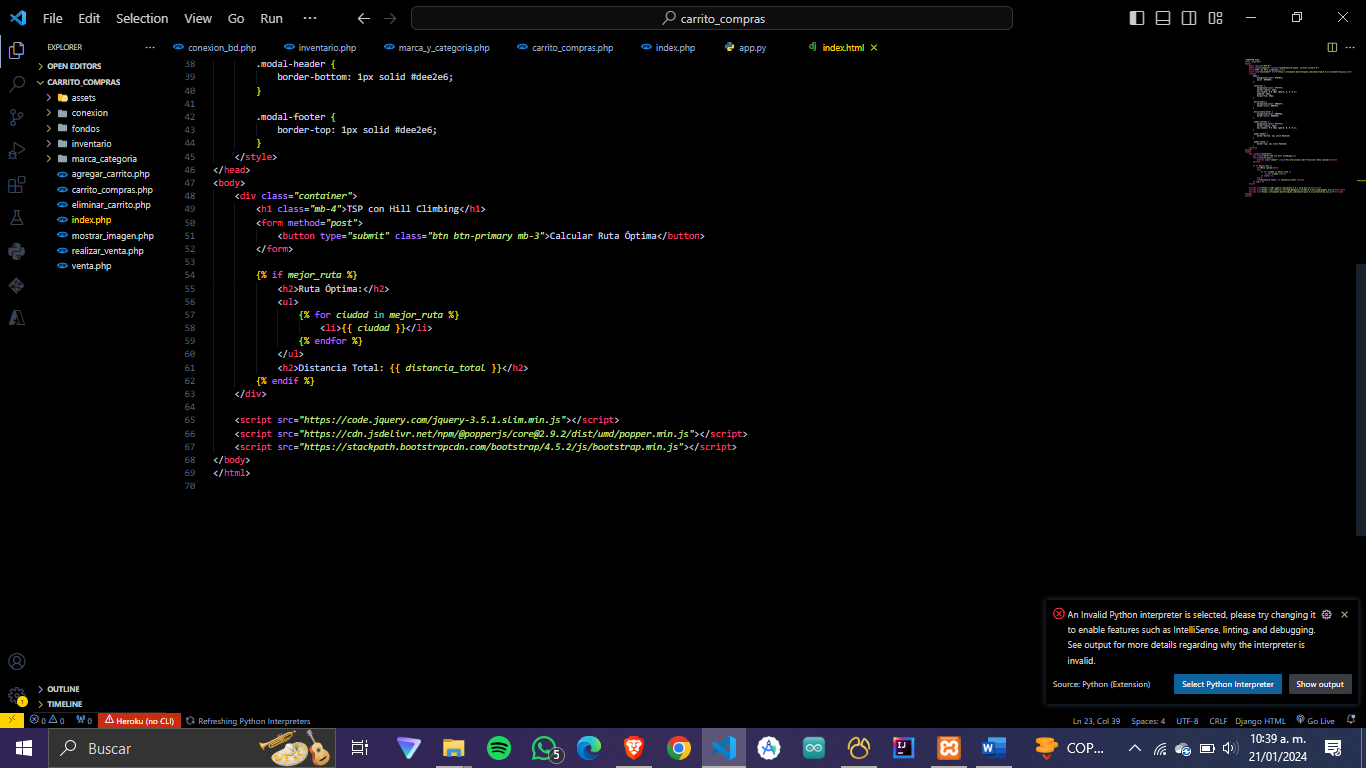
Y como es aplicación web igual tiene su index

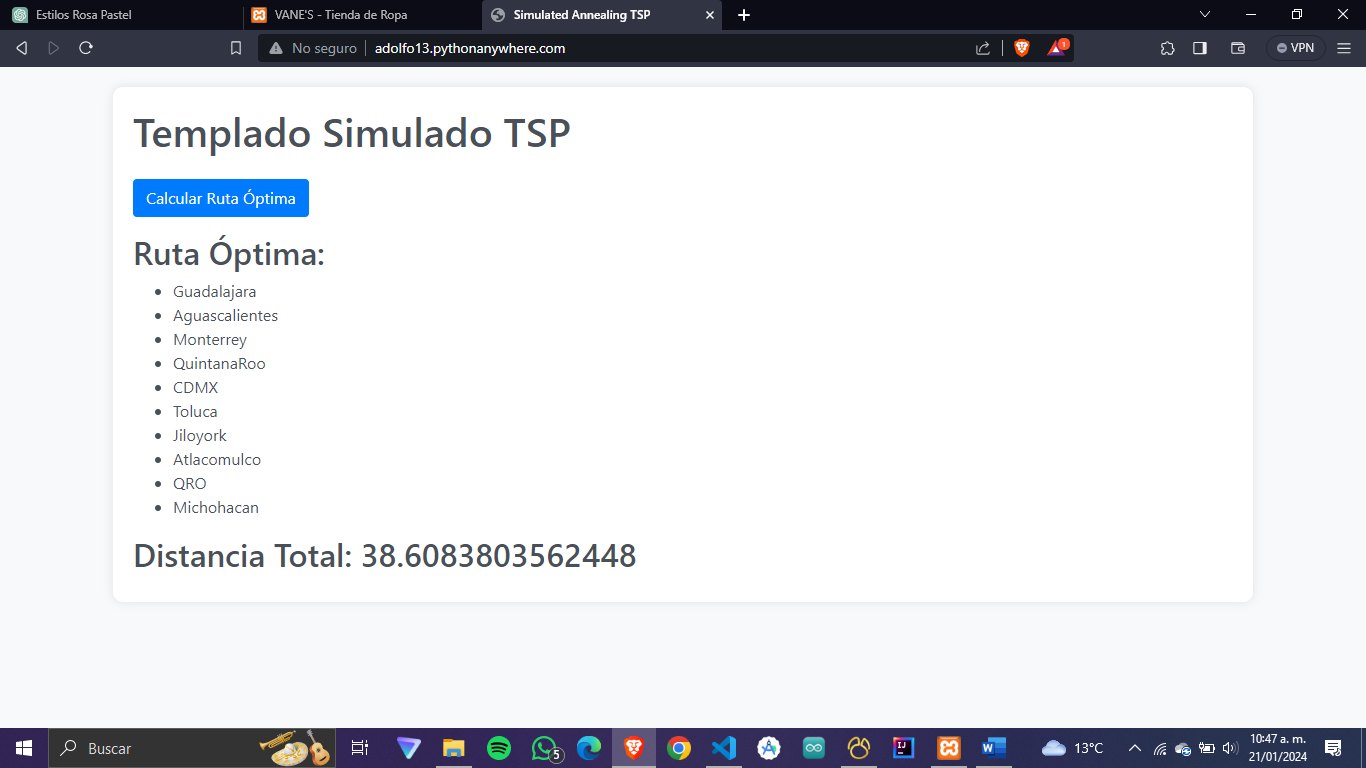


* 1. ALGORITMO DE HILL CLIMBING CON ITERACIONES   
     

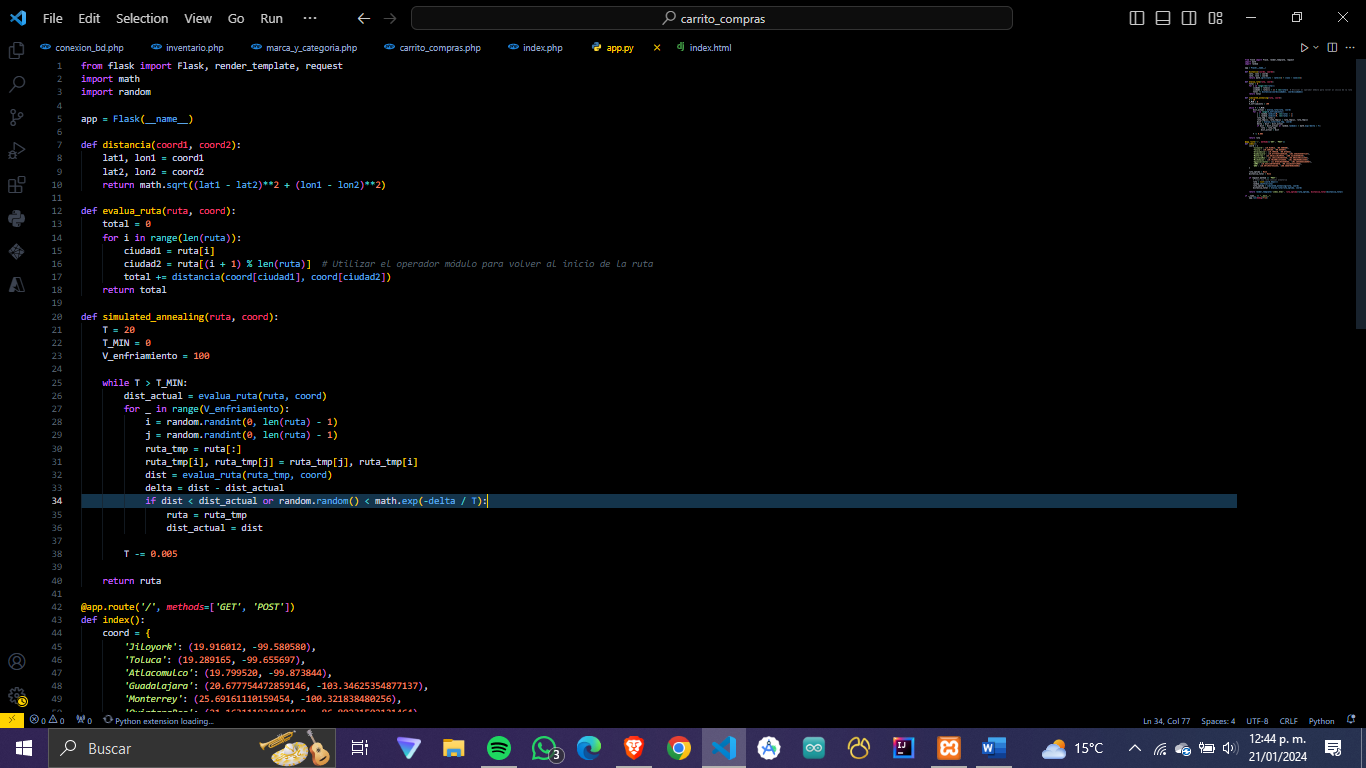
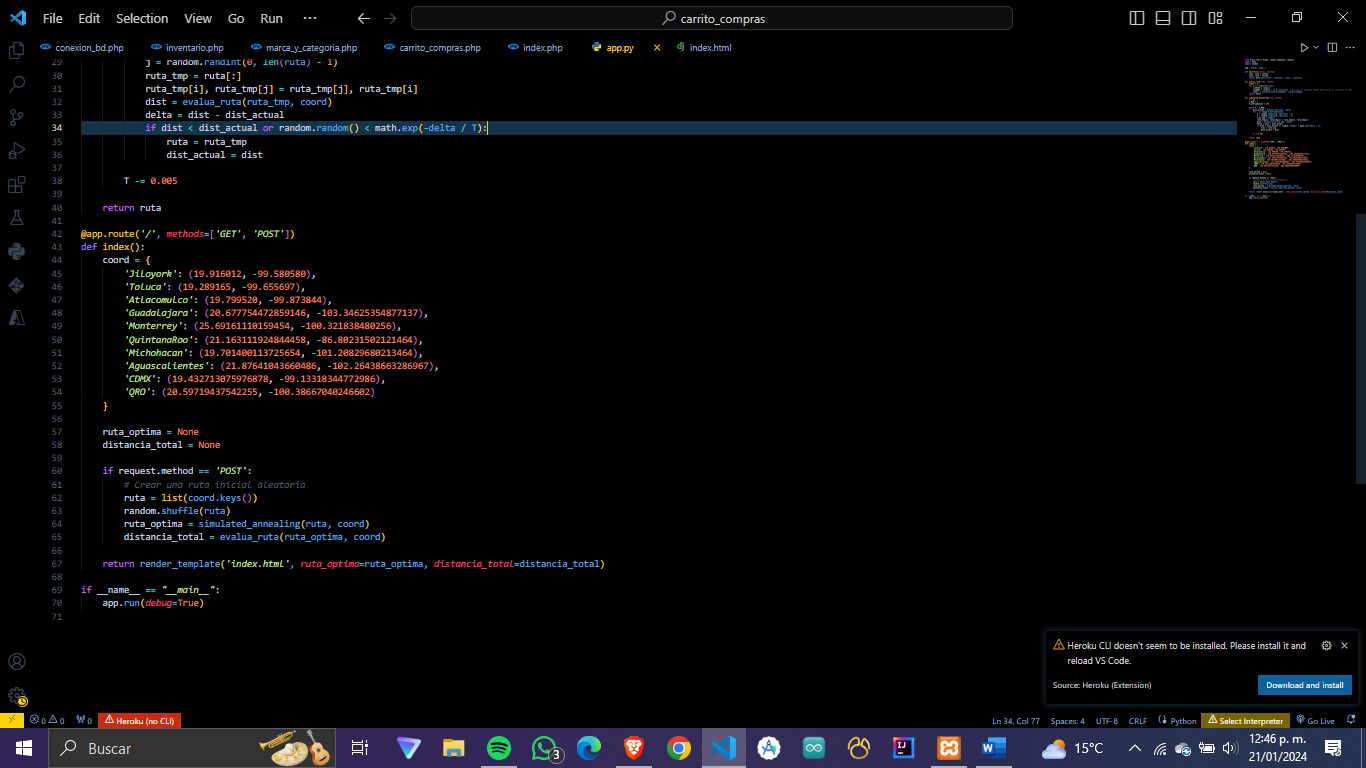
Este algoritmo hace iteraciones con cada ciudad y hace una distancio total de ellas para asi tener unas rutas mas optima de ellas y poder llegar a los puntos con menor distancia

Al igual que esta aplicación esta su index.html  


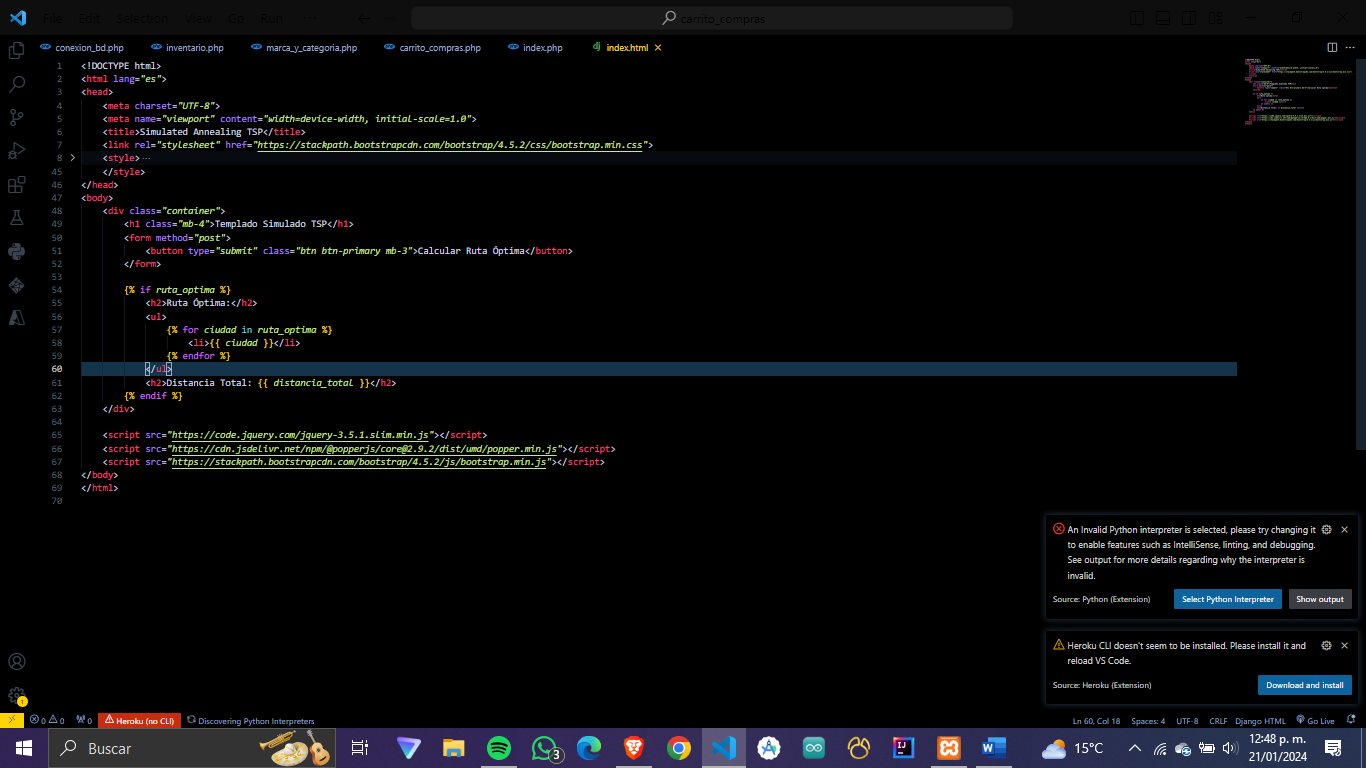
* 1. ALGORITMO TEMPLADO SUMULADO   
     

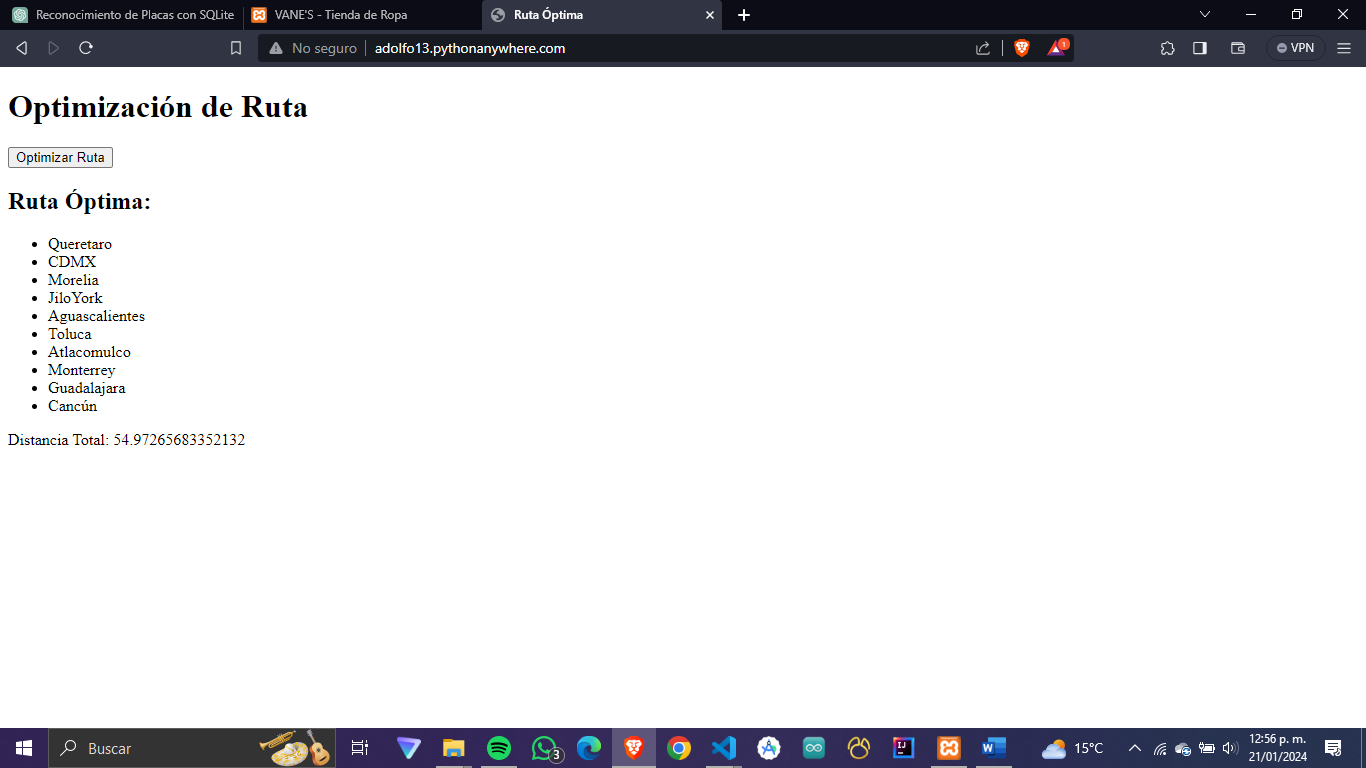
Este algotrimo lo hace es la distancia de cada ciudad una simulación de la distancia al igual que mejora en la forma de que busca la mejor opción de ciudades y la discancias de cada una de ellas

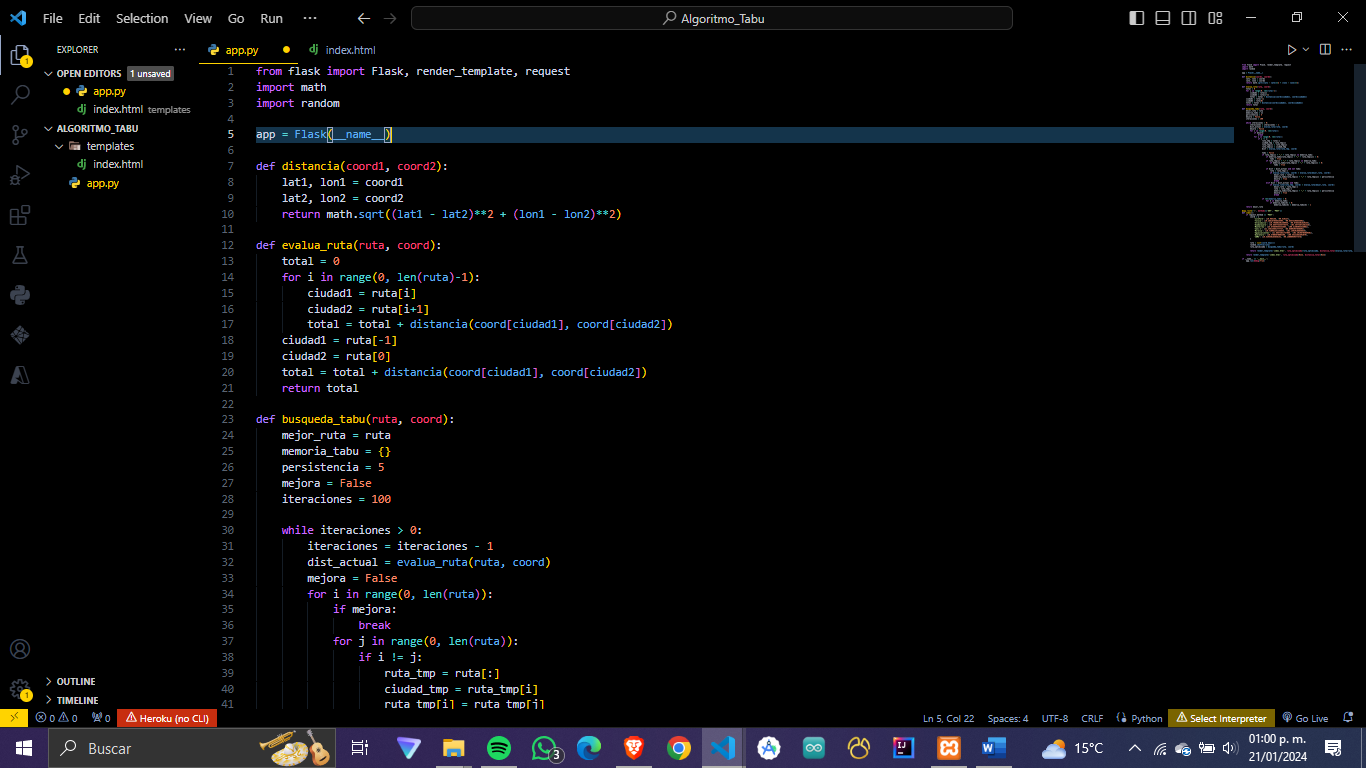
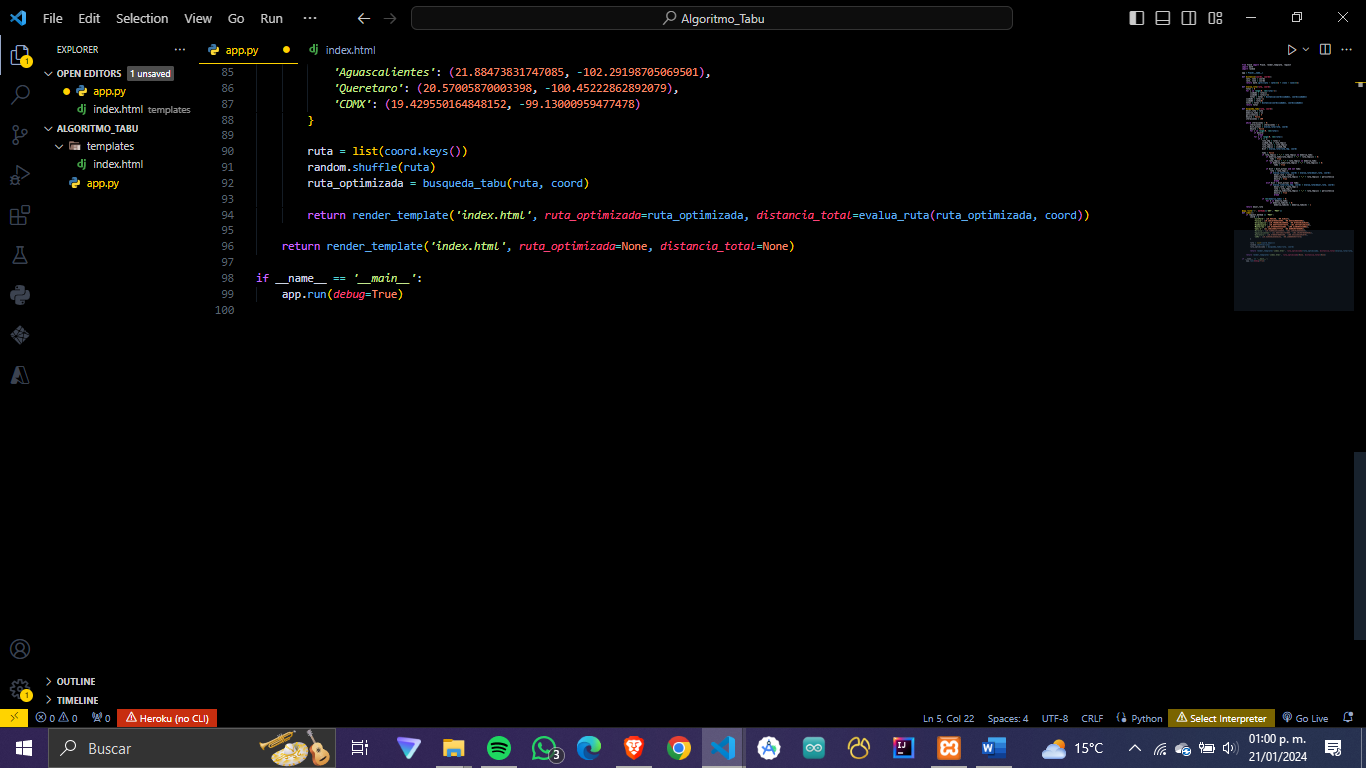
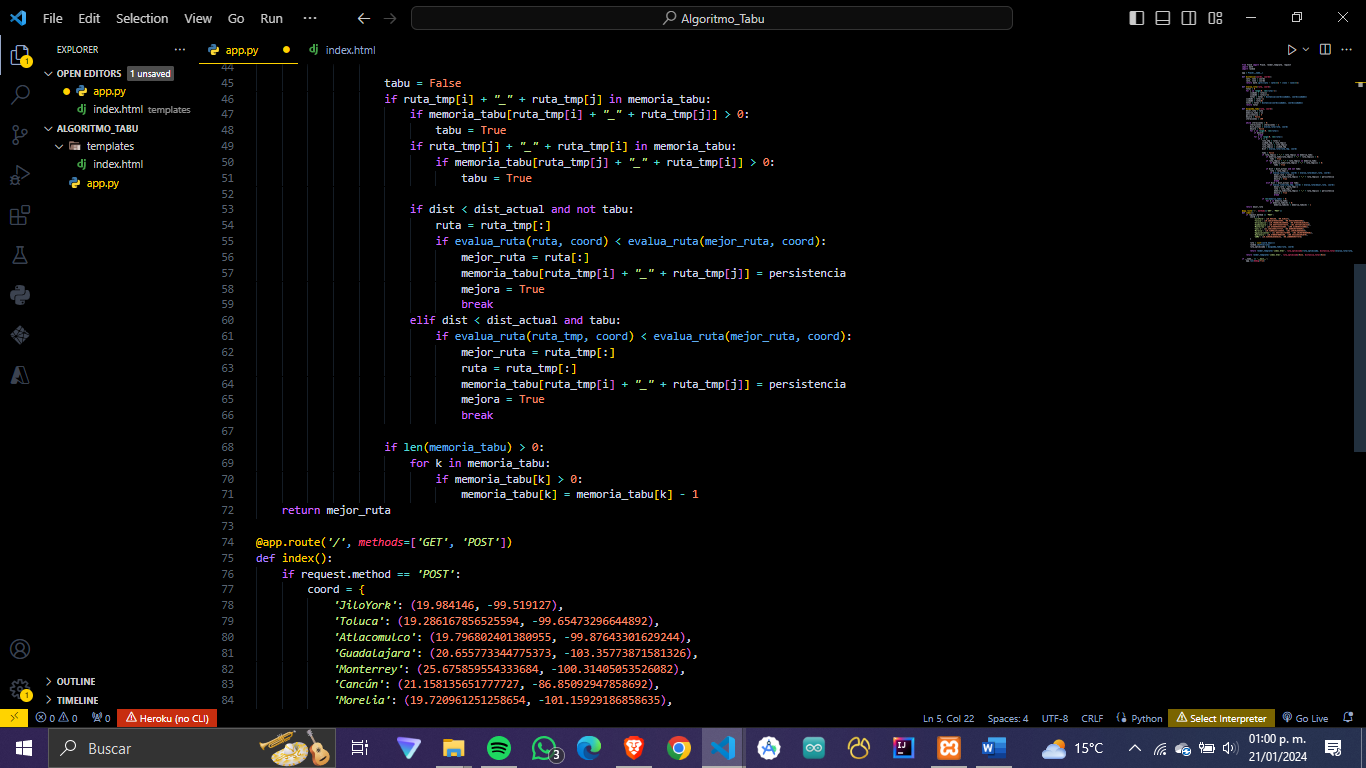
 

Y como es aolicacion web esta su index

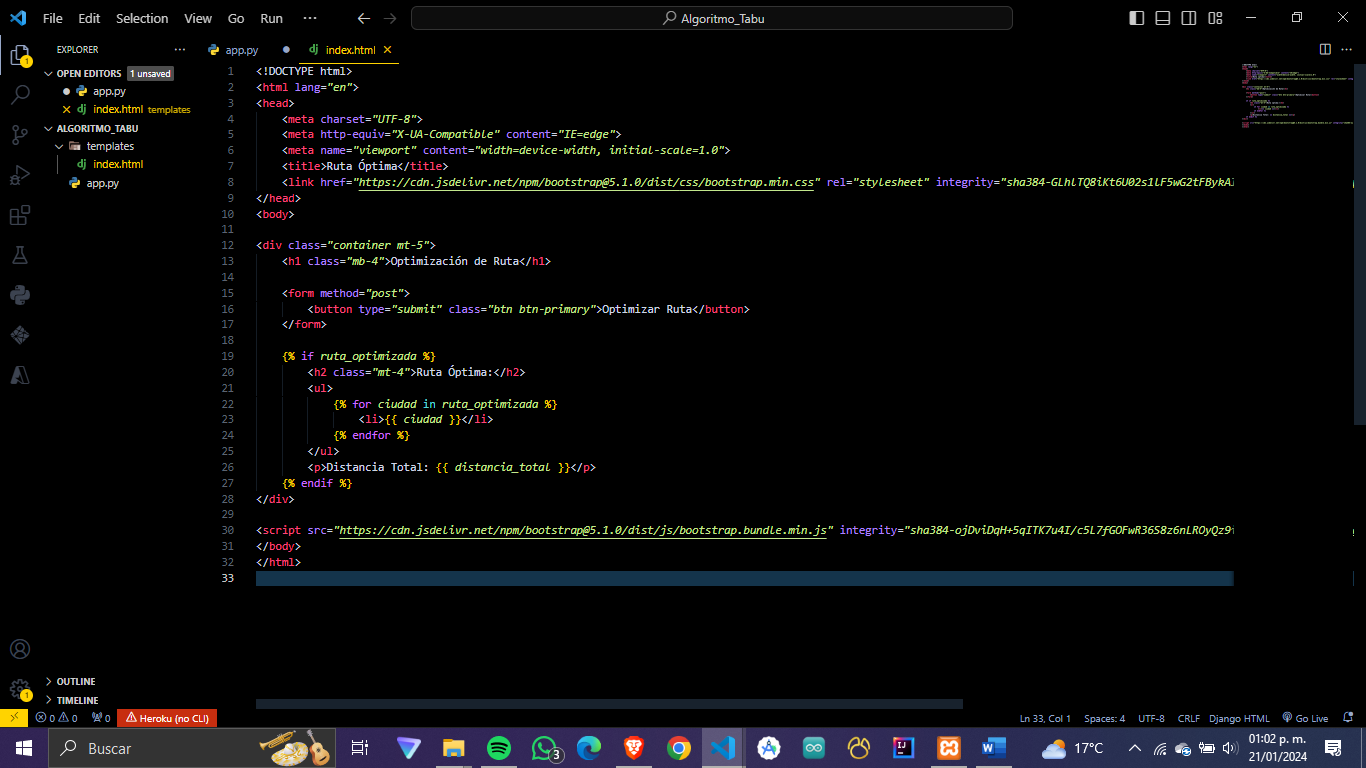
Recuerda que todo se sube tal cual en PythonAnywhere



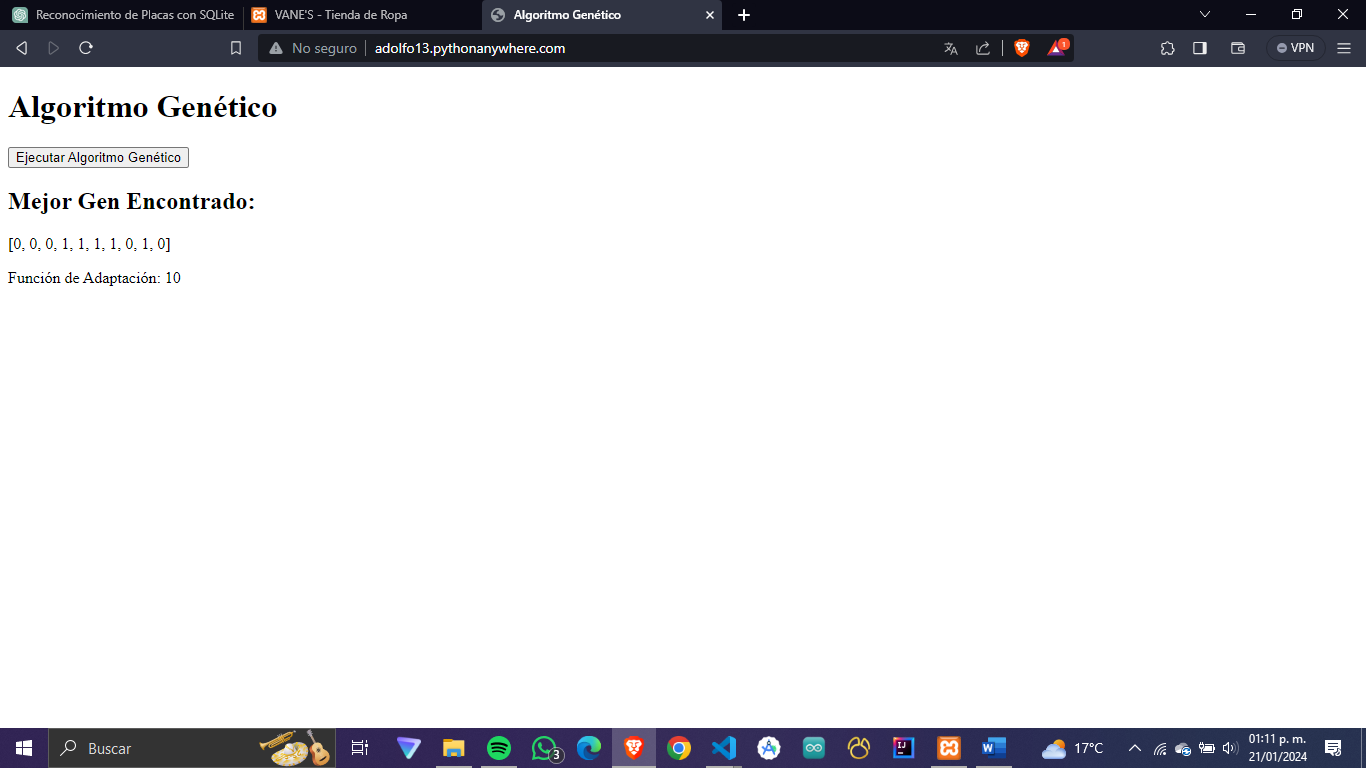
* 1. ALGORITMO BUSQUEDA TSP TABU  
     

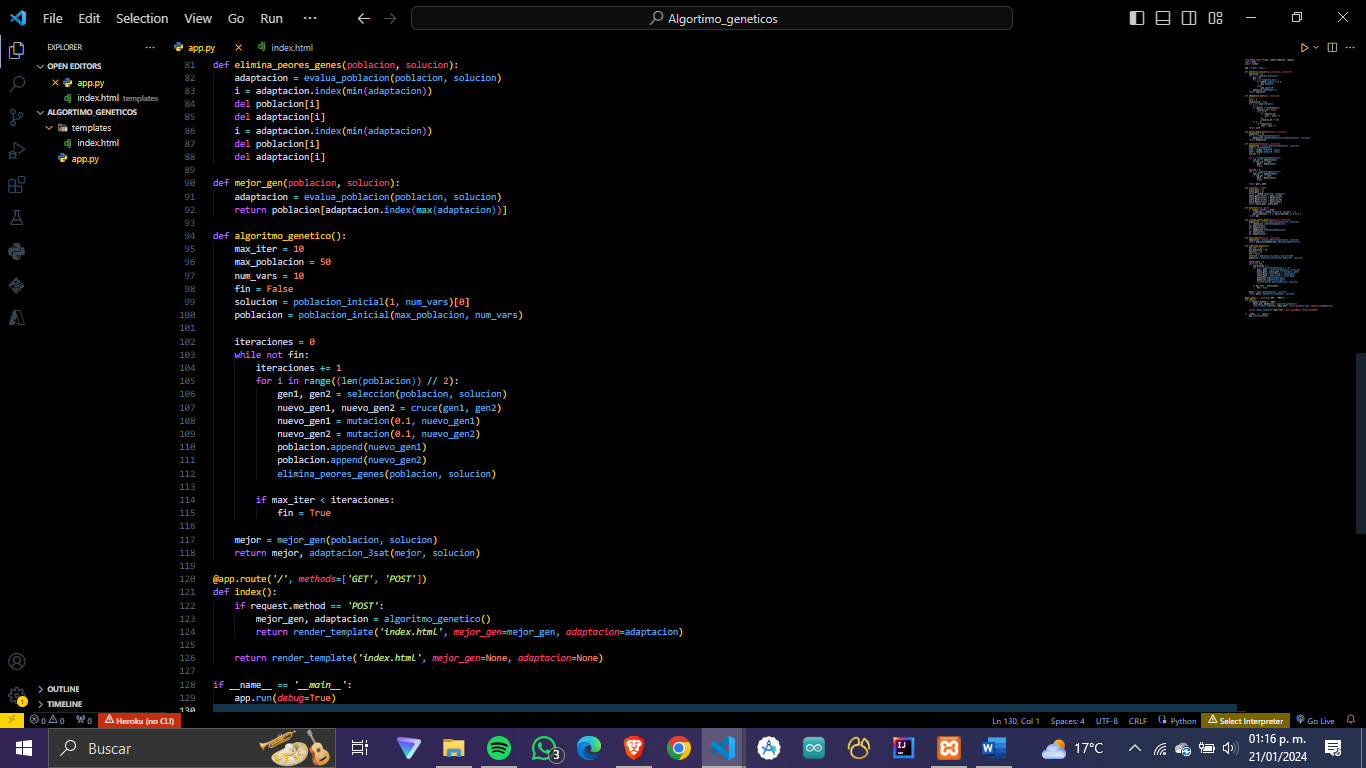
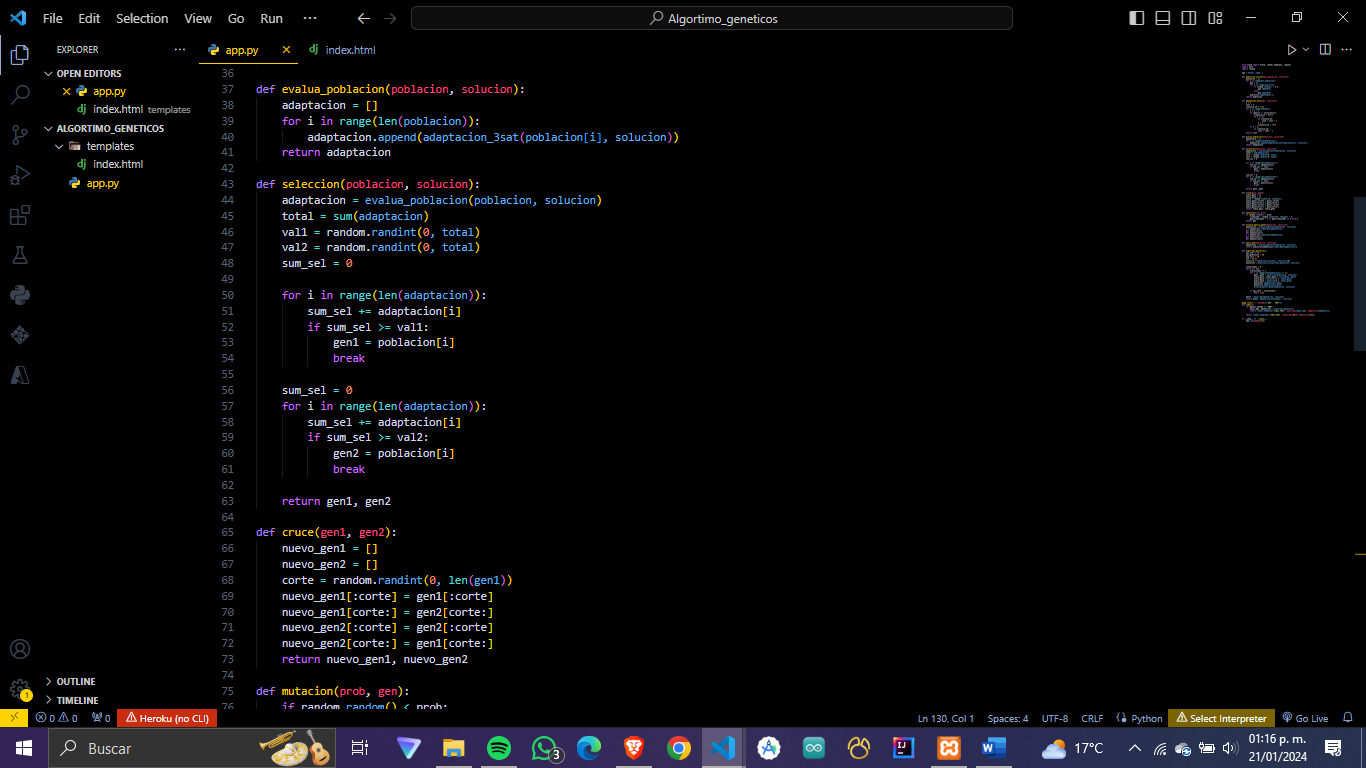
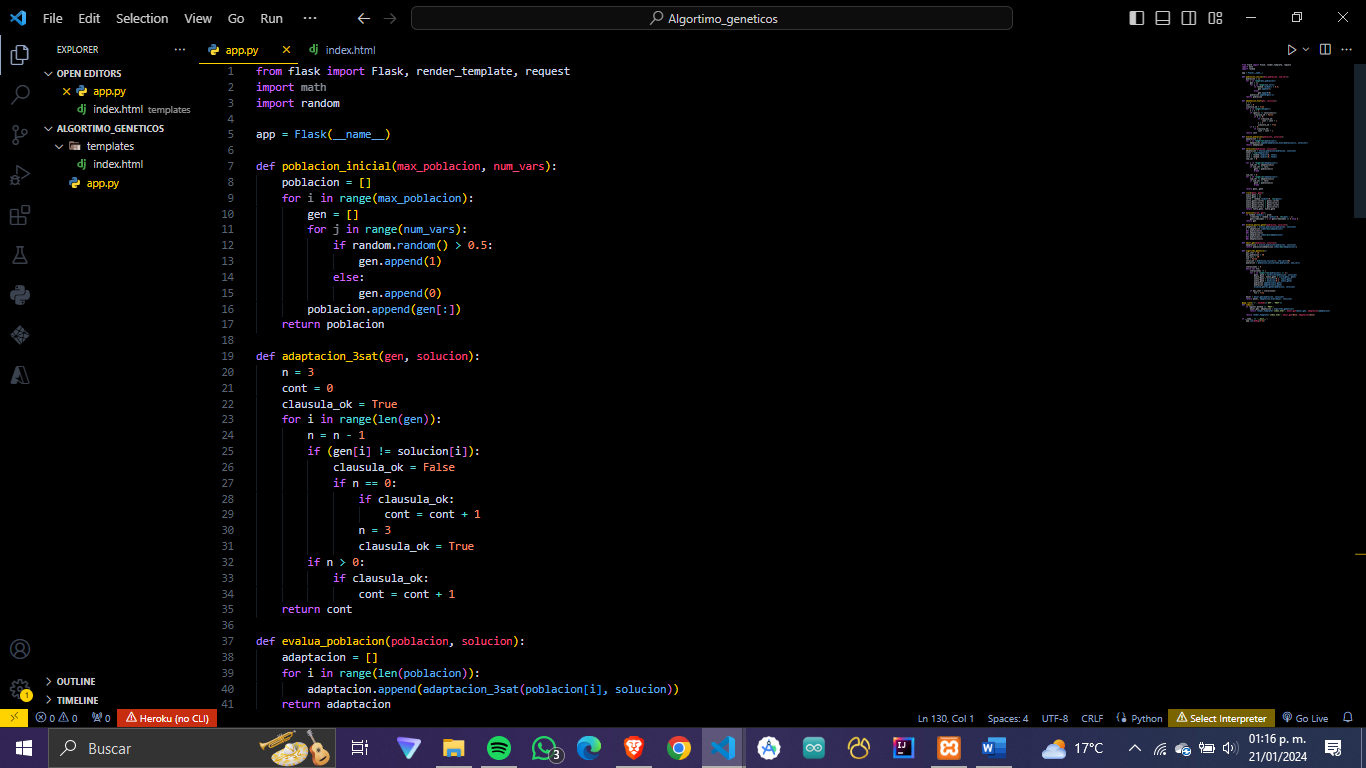
Este algoritmo hace una búsqueda en tabu es decir con el templado simulado este algoritmo hace una gran mejora xq muestra la distancia mas espesifica de cada cuidad a recorrer etse código lo que hcae es decir cuanto es la distancia estimada a llegar a una ciudad  
 

Y como es una aplicación web debe de tener su front para que tenga un mejor manejo y una visión agradable para el usuario



* 1. ALGORITMO GENETICOS



Lo que nos muestra es un gen encontrado podeis adaparlo asta 50 o mas cantidad de números lo que hace es buscar el mejor si lo aplicamos a lo real como se ha estado trabajando de las rutas lo que nos ara es presentar cierta cantidad de rutas y verificar cual es la mejor por comenzar  
asi que este algoritmo es interesante y te sacara de aprietos si asi lo deseas   
y este código se estructuro de la sigente manera   


Y como es aplicación web debe de tener su index.html  
