Modelos Bioinspirados y Heurísticas de Búsqueda

Practica 4

“Algoritmos de Optimización Basados en Colonias de Hormigas”

Antonio José Morano Moriña

Índice

1- Problema a resolver

El objetivo de la práctica es resolver el problema del TSP, en concreto vamos a evaluar los archivos ch130.tsp y a280.tsp, con los algoritmos basados en hormigas SH y SHE.

```
def leer_datos_tsp(nombre_archivo):
    coordenadas = []
    with open(nombre_archivo, 'r') as archivo:
        for _ in range(6):
            next(archivo)
        linea = archivo.readline().strip()
        while linea != 'EOF':
            valores = linea.split()
            x = float(valores[1])
            y = float(valores[2])
            coordenadas.append((x, y))
            linea = archivo.readline().strip()
    return coordenadas
CoordC130 = leer_datos_tsp(rutaC130)
CoordA280 = leer_datos_tsp(rutaA280)
```

leer_datos_tsp(nombre_archivo): Obtenemos la información del problema, las 6 primeras líneas las descartamos y cogemos el resto para obtener las coordenadas de cada ciudad y las devolvemos.

```
def DistanciaE(coord1, coord2):
    x1, y1 = coord1
    x2, y2 = coord2
    distancia = np.sqrt((x2 - x1)**2 + (y2 - y1)**2)
    return round(distancia)

def MatrizCostes(coordenadas):
    nCiudades = len(coordenadas)
    matriz_costes = np.zeros((nCiudades, nCiudades))

    for i in range(nCiudades):
        for j in range(nCiudades):
            if(i!=j):
                distancia = DistanciaE(coordenadas[i], coordenadas[j])
                matriz_costes[i][j] = distancia
            else:
                matriz_costes[i][j] = 0.0
    return matriz_costes

mCostesC130 = MatrizCostes(CoordC130)
mCostesA280 = MatrizCostes(CoordA280)
```

MatrizCostes(coordenadas): Genera una matriz de costes a partir de las coordenadas de las ciudades. Esta función recorre todas las ciudades y calcula la distancia euclidiana (DistanciaE) entre cada par de ciudades, almacenando los resultados en una matriz. La diagonal principal de la matriz se establece en 0.

```
def costeCamino(mCostes, camino):
    costeTotal = 0
    nCiudades = len(camino)

    for i in range(nCiudades):
        ciudadAct = camino[i]
        ciudadSig = camino[(i + 1) % nCiudades]

        coste = mCostes[ciudadAct][ciudadSig]
        costeTotal += coste

    return costeTotal
```

costeCamino(mCostes, camino): Calcula el coste total de un camino dado en función de una matriz de costes. La función itera sobre el camino y suma los costes entre cada par de ciudades consecutivas en la matriz de costes.

2-Algoritmo Greedy

```python
def solGreedy(coordenadas):
    nCiudades = len(coordenadas)
    visitados = [False] * nCiudades
    camino = []
    ciudadAct = 0
    camino.append(ciudadAct)
    visitados[ciudadAct] = True

    # Convertir las coordenadas en arrays de NumPy
    coordenadas = np.array(coordenadas)

    # Construir el camino greedy
    while len(camino) < nCiudades:
        dMin = float('inf')
        ciudadSig = -1

        # Encontrar la ciudad no visitada más cercana a la ciudad actual
        for i in range(nCiudades):
            if not visitados[i]:
                d = np.linalg.norm(coordenadas[ciudadAct] - coordenadas[i])
                if d < dMin:
                    dMin = d
                    ciudadSig = i

        # Marcar la ciudad como visitada y agregarla al camino
        visitados[ciudadSig] = True
        camino.append(ciudadSig)
        ciudadAct = ciudadSig

    return camino
```

solGreedy(coordenadas): Implementa una solución inicial greedy para el problema del TSP. Comienza desde una ciudad inicial y, en cada paso, elige la ciudad no visitada más cercana como siguiente ciudad en el camino. La función devuelve el camino obtenido.

3- Funciones auxiliares SH y SHE

```python
def ProbabilidadesCiudades(matrizDistancias, feromonas, solucion_actual, por_visitar, alpha, beta):
    ciudad_actual = solucion_actual[-1]
    probabilidades = []


    for ciudad in por_visitar:
        feromona = feromonas[ciudad_actual, ciudad]
        distancia = matrizDistancias[ciudad_actual, ciudad]

        if distancia == 0:
            probabilidad = 0.0
        else:
            probabilidad = (feromona ** alpha) * ((1.0 / distancia) ** beta)

        probabilidades.append(probabilidad)

    suma_probabilidades = sum(probabilidades)

    if suma_probabilidades == 0:
        probabilidades_normalizadas = [1.0 / len(probabilidades)] * len(probabilidades)  # Asignar u
    else:
        probabilidades_normalizadas = [p / suma_probabilidades for p in probabilidades]

    return probabilidades_normalizadas
```

ProbabilidadesCiudades(matrizDistancias, feromonas, solucion_actual, por_visitar, alpha, beta): Calcula las probabilidades de selección para las ciudades no visitadas en un momento dado durante la construcción del camino. Las probabilidades se calculan en función de las feromonas y las distancias entre la ciudad actual y las ciudades no visitadas.

```python
def SimulaCamino(matrizDistancias, feromonas, alpha, beta):
    nCiudades = matrizDistancias.shape[0]
    porVisitar = set(range(nCiudades))
    solucion_actual = [random.choice(list(porVisitar))]  # Convertimos el conjunto a una lista para poder hacer una selección al

    while porVisitar:
        probabilidad_transicion = ProbabilidadesCiudades(matrizDistancias, feromonas, solucion_actual, porVisitar, alpha, beta)
        siguiente_ciudad = random.choices(list(porVisitar), weights=probabilidad_transicion)[0]  # Seleccionamos la siguiente ci

        solucion_actual.append(siguiente_ciudad)
        porVisitar.remove(siguiente_ciudad)

    return solucion_actual
```

SimulaCamino(matrizDistancias, feromonas, alpha, beta): Simula la construcción de un camino utilizando las feromonas y las probabilidades de selección de las ciudades no visitadas. En cada paso, se selecciona aleatoriamente la siguiente ciudad de acuerdo con las probabilidades y se agrega al camino. La función devuelve el camino completo.

```python
def ActualizarFeromonas(feromonas, soluciones, costes, rho, Q):
    feromonas *= (1 - rho)  # Evaporación global de feromonas

    for solucion, coste in zip(soluciones, costes):
        for i in range(len(solucion) - 1):
            ciudad_actual = solucion[i]
            ciudadSig = solucion[i + 1]
            feromonas[ciudad_actual, ciudadSig] += Q / coste
            feromonas[ciudadSig, ciudad_actual] += Q / coste
```

ActualizarFeromonas(feromonas, soluciones, costes, rho, Q): Actualiza las feromonas de acuerdo con las soluciones generadas por las hormigas. Se aplica la evaporación global de feromonas y se deposita una cantidad de feromonas proporcional al inverso del coste de cada solución.

4-Algoritmo SH

```python
def tspACO(matrizDistancias, nHormigas, tMax, alpha, beta, rho, Q, problema):
    nCiudades = matrizDistancias.shape[0]
    if problema == 0:
        caminoGreedy = solGreedy(CoordC130)
        L = costeCamino(mCostesC130,caminoGreedy)
    else:
        caminoGreedy = solGreedy(CoordA280)
        L = costeCamino(mCostesA280,caminoGreedy)

    nEvals = 0
    feromonas = np.ones((nCiudades, nCiudades)) * (1 / (nCiudades * L ))
    solMejor = None
    costeMejor = np.inf
    tIni = time.time()
    mejorCoste = []

    while (time.time() - tIni) < tMax:
        soluciones = []
        costes = []

        for _ in range(nHormigas):
            solucion = SimulaCamino(matrizDistancias, feromonas, alpha, beta)
            coste = costeCamino(matrizDistancias, solucion)
            nEvals +=1

            if coste < costeMejor:
                solMejor = solucion
                costeMejor = coste

            mejorCoste.append(costeMejor)
            soluciones.append(solucion)
            costes.append(coste)

        ActualizarFeromonas(feromonas, soluciones, costes, rho, Q)

    return solMejor, costeMejor ,nEvals , mejorCoste
```

Esta función implementa el algoritmo SH para resolver el problema del TSP. Recibe una matriz de distancias entre ciudades, el número de hormigas, el tiempo máximo de ejecución, los parámetros de control del algoritmo (alpha, beta, rho, Q) y un indicador del problema. El algoritmo utiliza feromonas y una solución inicial greedy. Actualiza las feromonas según las soluciones generadas por las hormigas. Devuelve la mejor solución encontrada, su coste, el número de evaluaciones y una lista de los mejores costes en cada iteración.

5-Algoritmo SHE

```python
def tspACO_elitista(matrizDistancias, nHormigas, tMax, alpha, beta, rho, Q, nElite,problema):
    nCiudades = matrizDistancias.shape[0]
    if problema == 0:
        caminoGreedy = solGreedy(CoordC130)
        L = costeCamino(mCostesC130,caminoGreedy)
    else:
        caminoGreedy = solGreedy(CoordA280)
        L = costeCamino(mCostesA280,caminoGreedy)

    feromonas = np.ones((nCiudades, nCiudades)) * (1 / (nCiudades * L ))
    solMejor = None
    costeMejor = np.inf
    tIni = time.time()
    nEvals = 0
    mejoresCostes = []

    while (time.time() - tIni) < tMax:
        soluciones = []
        costos = []

        for _ in range(nHormigas):
            solucion = SimulaCamino(matrizDistancias, feromonas, alpha, beta)
            costo = costeCamino(matrizDistancias, solucion)
            nEvals+=1

            if costo < costeMejor:
                solMejor = solucion
                costeMejor = costo

            mejoresCostes.append(costeMejor)
            soluciones.append(solucion)
            costos.append(costo)

        # ActualizarFeromonas(feromonas, soluciones, costos, rho, Q)
        elitismo = soluciones
        elitismo[0:nElite] = SolucionesElite(soluciones, costos, nElite)
        ActualizarFeromonas(feromonas, elitismo, costos, rho, Q)

    return solMejor, costeMejor, nEvals , mejoresCostes
```

El algoritmo SHE introduce un nuevo parámetro llamado `nElite`, que indica la cantidad de soluciones elite que se seleccionarán en cada iteración. Estas soluciones elite se utilizarán para actualizar las feromonas.

La diferencia principal entre el algoritmo SHE y el algoritmo SH radica en la inclusión de una estrategia de elitismo en el SHE. Esta estrategia selecciona las mejores soluciones generadas por las hormigas en cada iteración y las utiliza para actualizar las feromonas. Esto puede ayudar a acelerar la convergencia y mejorar la calidad de las soluciones obtenidas.

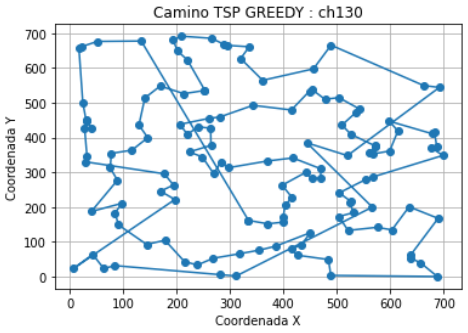
```python
def SolucionesElite(soluciones, costos, nElite):
    orden = np.argsort(costos)
    elitismo = [soluciones[i] for i in orden[:nElite]]
    return elitismo
```

SolucionesElite(soluciones, costos, nElite): Esta función selecciona la elite de las soluciones obtenidas, ordenándolas en orden ascendente y seleccionando las primeras, después las devuelve.

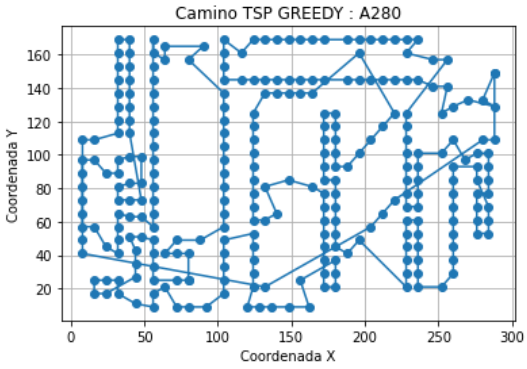
6- Experimentación caminos encontrados y evolución de los costes

A continuación, se muestran las imágenes de los mejores caminos obtenidos por cada uno de los algoritmos, durante cada ejecución, junto a la evolución que ha tenido el mejor coste durante la misma:

Greedy



Problema ch130



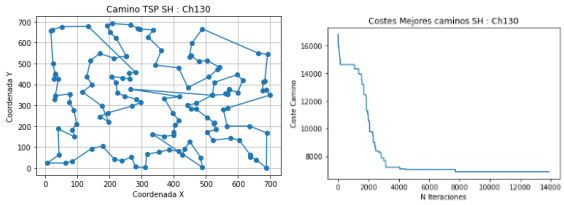
Problema a280

Las graficas correspondientes a las semillas 10 y 200, están resaltadas en rojo, ya que en ellas se estudia cómo evoluciona el mejorCoste si no inicializamos las feromonas iniciales sembrando la solución greedy. En estas podemos notar claramente como los algoritmos necesitan explorar mucho más el espacio para encontrar soluciones optimas, ya que no tiene ningún tipo de ajuste inicial en las feromonas, haciendo que se note mucho el tiempo que tarda en descender los costes en las gráficas con respecto a las ejecuciones donde si se sembraron inicialmente las feromonas correspondientes a la solución para el problema encontrada por el algoritmo greedy. Mas allá de eso, suelen llegar a soluciones similares.

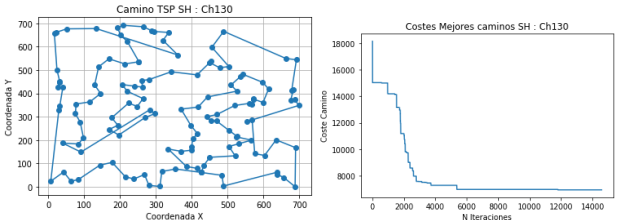
Los resultados almacenados en la tabla son todos obtenidos sembrando la solución greedy inicialmente, como se pide en el enunciado de la práctica. Esto es una experimentación aparte.

Algoritmo SH

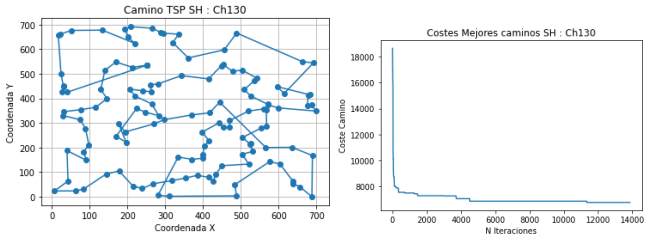
Problema ch130



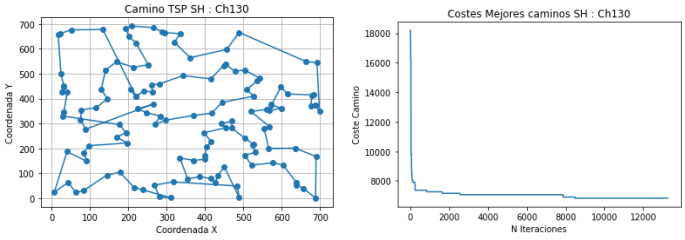
Semilla 10



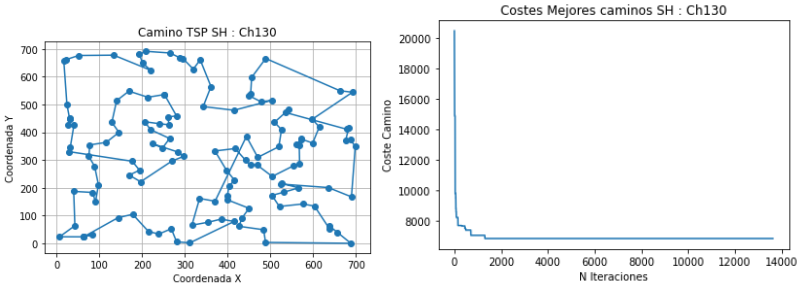
Semilla 200



Semilla 15

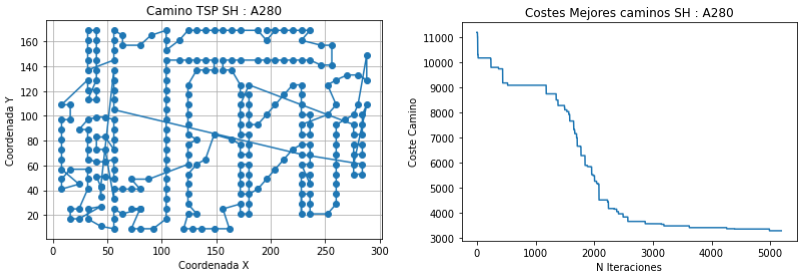


Semilla 50

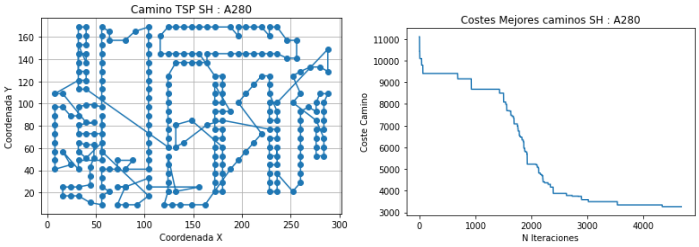


Semilla 100

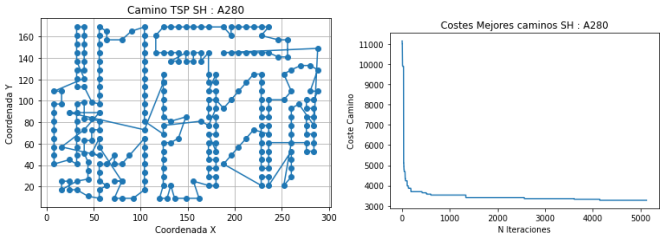
Problema a280



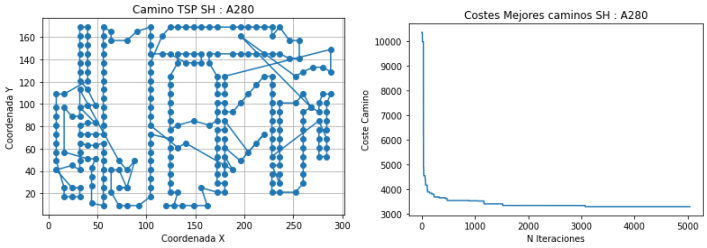
Semilla 10



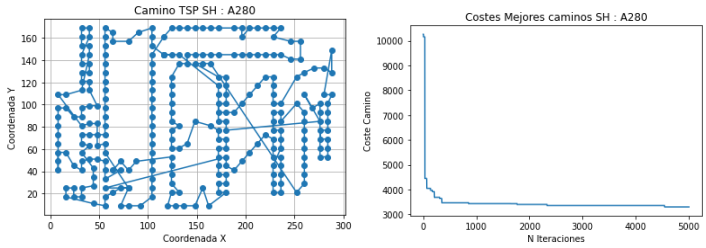
Semilla 200



Semilla 15



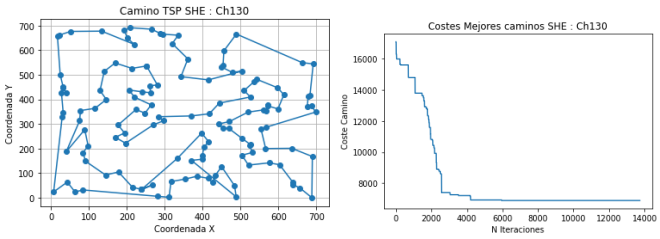
Semilla 50



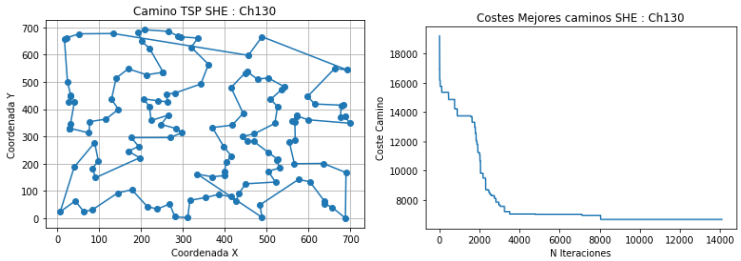
Semilla 100

Algoritmo SHE

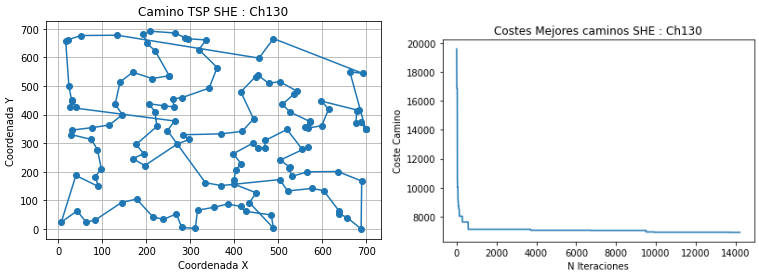
Problema ch130



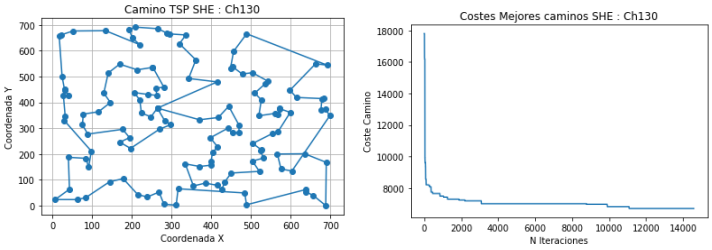
Semilla 10



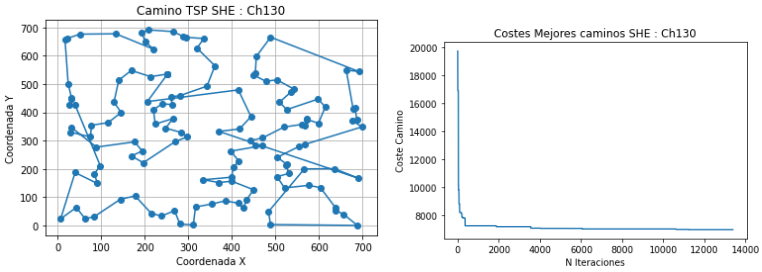
Semilla 200



Semilla 15

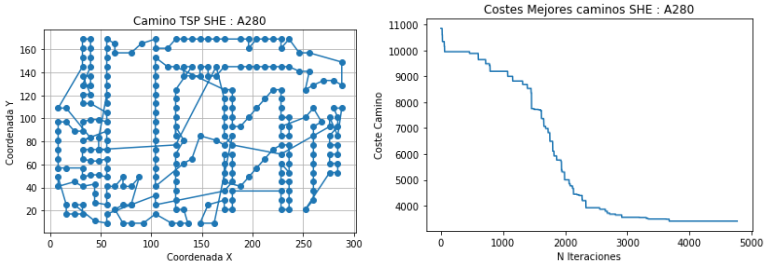


Semilla 50

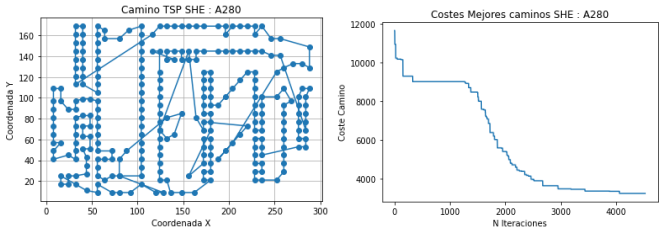


Semilla 100

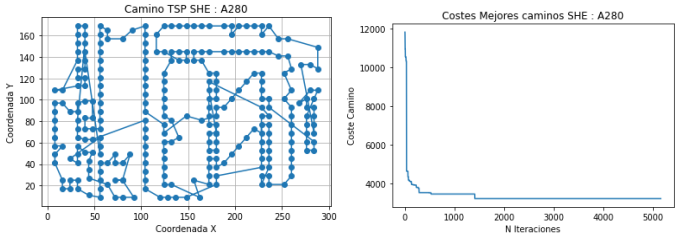
Problema a280



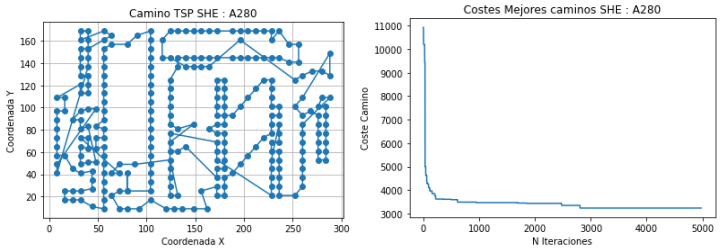
Semilla 10



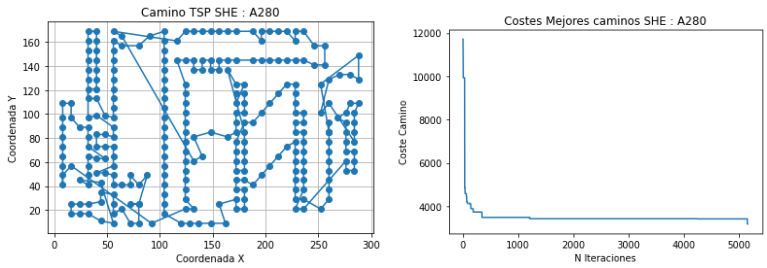
Semilla 200



Semilla 15



Semilla 50



Semilla 100

Observando los caminos obtenidos durante las diferentes ejecuciones, podemos ver que estos son muy parecidos entre sí, aunque presentan diferencias en algunas ciudades visitadas, lo que ocasiona los pequeños cambios de costes que hay entre cada una de las soluciones, aun así, todas tienden a seguir un determinado patrón, que será el más optimo que es capaz de encontrar los algoritmos durante el tiempo de ejecución que les es permitido.

En cuanto a la evolución de los costes, más allá de lo antes comentado sobre la experimentación de la evolución del mejor coste, podemos ver como todos tienden a alcanzar soluciones optimas con mucha facilidad, en comparación con el coste computacional que le supone salir de estos óptimos locales en los que se queda, lo que hace que no encuentren el óptimo global del problema. Por lo que creo que, si se dejara mas tiempo a la ejecución de los algoritmos o se ejecutaran en máquinas más potentes, es probable que durante alguna de las iteraciones se encontrara el optimo global de los problemas propuestos.

7-Resultados

Resultados para el problema Ch130.tsp

	GREEDY(coste/evals)		SH(coste/evals)		SHE(coste/evals)	
Ejecución 1	7578	1	6884	13890	6787	13740
Ejecución 2	7578	1	6821	13860	6694	13950
Ejecución 3	7578	1	6868	13050	6986	13050
Ejecución 4	7578	1	6761	12930	6912	13200
Ejecución 5	7578	1	6937	14580	6669	14100
Media	7578	1	6854,2	13662	6829,6	13608
Desv. Típica	-	-	66,56	679,02	140,29	462,13

En la tabla de resultados podemos observar como el algoritmo que siempre encuentra el peor camino es el greedy, como es normal en un principio.

Si observamos los algoritmos basados en hormigas, podemos observar que tampoco existe una gran diferencia entre los rendimientos de cada uno, el algoritmo SHE es capaz de encontrar la solución mas cercana al optimo global de todas las ejecuciones, aun así la media de las soluciones de cada uno de los algoritmos es muy parecida, al igual que el coste computacional, esto se debe a que tampoco son algoritmos que presenten muchas diferencias en su implementación, lo que se ve reflejado en los valores de la tabla.

Podríamos resumir con que el algoritmo SH es mas robusto tanto en cantidad de evaluaciones como en costes encontrados, aun así, en ambas su media es algo peor que la del SHE. Encontrando ambos algoritmos mejores resultados que la solución Greedy.

Resultados para el problema A280.tsp

	GREEDY (coste/evals)		SH (coste/evals)		SHE (coste/evals)	
Ejecución 1	3139	1	3285	8490	3233	8220
Ejecución 2	3139	1	3271	8610	3229	8050
Ejecución 3	3139	1	3302	8550	3176	8310
Ejecución 4	3139	1	3235	8310	3176	8460
Ejecución 5	3139	1	3303	8580	3247	8250
Media	3139	1	3279,2	8508	3212,2	8258
Desv. Típica	-	-	28,02	119,25	33,71	148,56

En este problema, los resultados de la tabla varían como era de esperar, aunque ahora el algoritmo Greedy, que es el más básico de todos, encuentra mejores soluciones.

No entiendo muy bien porque ya que los algoritmos basados en hormigas actualizan su matriz de feromonas de acuerdo con la solución inicial obtenido por el algoritmo Greedy, y aun así no encuentra resultados ni iguales de buenos a estos, tras iterar durante 8 minutos en busca de mejores soluciones.

Analizando los resultados obtenidos, podemos decir a parte del cambio en los resultados de la solución greedy, los algoritmos basaos en hormigas se siguen comportando de forma parecida entre sí, ya que al igual que en el problema anterior, los mejores costes entre estos son encontrados por el SHE, pero este vuelve a ser menos robusto tanto en resultados como en evaluaciones efectuadas con respecto al SH.

Las medias tanto de costes como de numero de llamadas a la función de evaluación también son prácticamente iguales para ambos algoritmos, volviendo a demostrar la poca diferencia que existe entre estos.

8-Bibliografía científica

Peñuela, C. A., Franco, J. F., & Toro, E. M. (2008). Colonia de hormigas aplicada a la programación óptima de horarios de clase. *Scientia et technica*, *1*(38).

Se explica el problema de la programación óptima de horarios de clase, junto con el modelo matemático que lo representa y su respectiva codificación. Se muestra la aplicación de la Colonia de Hormigas en la solución del problema.

Obando Solano, J. P., Zamora Moreno, J. A., & Giraldo Ramos, F. N. (2016). Algoritmo de optimización de colonias de hormigas para el problema de distribución en planta.

En este trabajo se busca la optimización a problemas propios del campo de distribución en plantas, aplicando inteligencia artificial de enjambres, a partir de la implementación de un algoritmo de Optimización de Colonias de Hormigas (Ant Colony Optimization - ACO)

Salazar Hornig, E., & Ruiz Fuentealba, N. (2009). Modelo ACO para la recolección de residuos por contenedores. *Ingeniare. Revista chilena de ingeniería*, *17*(2), 236-243.

En este trabajo se resuelve el Problema de Recolección de Residuos Domiciliarios por Contenedores, el que aplica un concepto de secuencias parciales de recolección que deben ser unidas para minimizar la distancia total de recolección. El problema de unir las secuencias parciales se representa como un TSP, el que es resuelto mediante un algoritmo ACO.

Arito, F. L. A. (2010). Algoritmos de Optimización basados en Colonias de Hormigas aplicados al Problema de Asignación Cuadrática y otros problemas relacionados. *Universidad Nacional de San Luis Facultad de Ciencias Físico Matemáticas y Naturales Departamento de Informática, San Luis–Argentina Abril de*.2010

En esta tesis se explica el problema de asignación cuadrática, tras esto se aplica a este problema varios algoritmos de Ant Colony Optimization, con pequeños cambios entre ellos y se analizan los resultados.