Universidad de Huelva

Universidad de Huelva

Grado en Ingeniería Informática

Práctica 4: Algoritmos de Optimización Basados en Colonia de Hormigas

Memoria de Prácticas

Autor: Alberto Fernández Merchán
Asignatura: Modelos Bioinspirados y Heurística de Búsqueda
Profesor: Miguel Ángel Rodríguez Román

Índice

1. Introducción **3**

2. Greedy **4**
2.1. Introducción . . . 4
2.2. Análisis del algoritmo . . . 4
2.3. Código . . . 5
2.4. Resultados . . . 5
2.4.1. CH130 . . . 5
2.4.2. A280 . . . 6

3. Sistema de Hormigas **7**
3.1. Introducción . . . 7
3.2. Análisis del algoritmo . . . 7
3.3. Código . . . 8
3.3.1. Actualización de Feromonas . . . 8
3.3.2. Función de Transición . . . 8
3.3.3. Sistema de Hormigas . . . 8
3.4. Resultados . . . 9
3.4.1. CH130 . . . 9
3.4.2. A280 . . . 10

4. Sistema de Hormigas Elitista **11**
4.1. Introducción . . . 11
4.2. Análisis del algoritmo . . . 11
4.3. Código . . . 12
4.3.1. Actualización de Feromonas . . . 12
4.4. Resultados . . . 13
4.4.1. CH130 . . . 13
4.4.2. A280 . . . 13

5. Resultados **14**
5.1. CH130 . . . 14
5.2. A280 . . . 15

6. Conclusiones **15**

Índice de cuadros

1. Semillas 3
2. Resultados de Sistema de Hormigas sobre CH130 9
3. Resultados de Sistema de Hormigas sobre A280 10
4. Resultados obtenidos con el algoritmo de Sistema de Hormigas Elitista para el problema ch130. 13
5. Resultados obtenidos con el algoritmo de Sistema de Hormigas Elitista para el problema a280. 13
6. Resultados para el problema CH130. 14
7. Resultados para el problema A280. 15

1. Introducción

En esta práctica estudiaremos los algoritmos de optimización basados en colonia de hormigas (OCH). Estos algoritmos se inspiran en el comportamiento de las hormigas reales en una colonia artificial, y son utilizados para resolver problemas complejos de camino mínimo. En particular, nos enfocaremos en los algoritmos de *Sistema de Hormigas* y *Sistema de Hormigas Elitista*, los cuales han demostrado ser eficaces en la resolución del problema del viajante de comercio (TSP).

El problema del viajante de comercio consiste en encontrar la ruta más corta que pasa por todos los nodos de un grafo, volviendo al nodo de partida. Para poner a prueba los algoritmos de OCH, utilizaremos los ficheros *ch130.tsp* y *a280.tsp*, que contienen información sobre las distancias entre los nodos en dos conjuntos de datos diferentes. Estudiaremos el rendimiento de los algoritmos utilizando cinco semillas distintas para cada uno, lo cual nos permitirá obtener resultados más robustos y realizar comparaciones significativas.

Durante el estudio, analizaremos diferentes métricas para evaluar los resultados obtenidos por los algoritmos de OCH, así como el número de llamadas a la función de coste de cada algoritmo en relación con cada problema. Esto nos brindará información sobre la eficiencia y calidad de las soluciones generadas por los algoritmos.

Además de los algoritmos de OCH, también incluiremos un algoritmo *Greedy* en nuestro análisis, con el fin de realizar una comparación directa entre este enfoque heurístico y los algoritmos basados en colonias de hormigas. La inclusión de un enfoque greedy nos permitirá evaluar si los algoritmos de OCH logran superar o mejorar las soluciones proporcionadas por este método más simple pero menos sofisticado.

Por último, revisaremos algunos artículos relevantes en los cuales se ha utilizado los algoritmos de OCH para resolver problemas reales. Esto nos brindará una visión más amplia de las aplicaciones prácticas y los beneficios de estos algoritmos en diversos campos, como la logística, la planificación de rutas y la optimización de redes, entre otros.

Las semillas que se utilizarán para el estudio de los algoritmos son las siguientes:

Semillas				
123456	789456	456123	369852	852321

Cuadro 1: Semillas que se han utilizado para estudiar el funcionamiento de los algoritmos.

2. Greedy

2.1. Introducción

Como primera aproximación al problema del viajante de comercio, implementaremos un algoritmo greedy para encontrar un camino de forma rápida. El enfoque greedy es una estrategia heurística que busca tomar decisiones locales óptimas en cada paso, con la esperanza de alcanzar una solución global aceptable. En el caso del problema del viajante de comercio, donde se busca encontrar la ruta más corta que pase por todos los nodos de un grafo, el algoritmo greedy construye el camino paso a paso eligiendo en cada momento el nodo más cercano al nodo actual.

El algoritmo comienza seleccionando un nodo de partida y lo marca como el nodo actual. Luego, en cada iteración, busca el nodo más cercano al nodo actual que aún no haya sido visitado y lo agrega al camino. Este proceso continúa hasta que todos los nodos han sido visitados y el camino completo ha sido construido. El algoritmo greedy garantiza que nunca se visite un nodo más de una vez y que al final se obtenga un circuito cerrado que recorre todos los nodos.

Aunque el algoritmo greedy ofrece una solución rápida, no garantiza encontrar siempre la solución óptima. En algunos casos, puede producir rutas subóptimas que son significativamente más largas que la ruta más corta posible. Sin embargo, debido a su eficiencia, el enfoque greedy se utiliza a menudo como punto de partida para encontrar soluciones aproximadas al problema del viajante de comercio antes de aplicar algoritmos más complejos y computacionalmente costosos, como puede ser el caso de los sistemas basados en colonias de hormigas.

2.2. Análisis del algoritmo

El algoritmo mantiene un *array* de ciudades visitadas para garantizar que cada ciudad sea visitada una sola vez. También mantiene un *array* que representa el camino que ha recorrido el viajero hasta el momento. En cada iteración, se actualiza el estado de las ciudades visitadas y el recorrido.

Aunque el algoritmo es relativamente simple de implementar, su eficiencia no es óptima. La búsqueda voraz puede conducir a soluciones subóptimas, ya que se toma la decisión óptima en cada paso sin considerar las posibles consecuencias a largo plazo. En el peor caso, el algoritmo puede requerir un tiempo cuadrático en función del número de ciudades.

Las ventajas que puede proporcionar este algoritmo pueden ser una fácil legilibidad e implementación, sin embargo, tiene un coste computacional de $O(n^2)$ y no garantiza la solución óptima al problema.

2.3. Código

```python
def tsp_greedy(coords):
    n = len(coords)
    visited = [False] * n
    path = [0]
    visited[0] = True
    current_city = 0
    while len(path) < n:
        min_distance = np.inf
        nearest_city = None
        for i in range(n):
            if not visited[i]:
                distance = Utils.distanciaEuclidea(coords[current_city], coords[i])
                if distance < min_distance:
                    min_distance = distance
                    nearest_city = i
        visited[nearest_city] = True
        path.append(nearest_city)
        current_city = nearest_city
    return path
```

2.4. Resultados

A continuación se muestran los resultados obtenidos en ambos problemas con el algoritmo greedy descrito anteriormente:

2.4.1. CH130

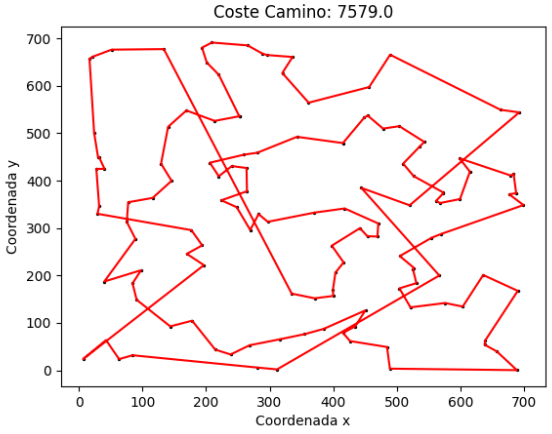


Figura 1: Resultados del algoritmo Greedy sobre el problema CH130

En la imagen anterior se muestra el camino generado por el algoritmo greedy implementado en esta sección para resolver el problema CH130.

En el caso del algoritmo voraz, el camino no mejora con las iteraciones, ya que en una sola iteración se genera el camino que tiene las aristas más cortas desde el punto en el que se encuentra. Debido a esta característica, no se ha generado ninguna gráfica que muestre la evolución del camino mínimo a lo largo de

las iteraciones.

Además, dado que el algoritmo no tiene componentes aleatorios, el camino greedy es siempre el mismo para todas las semillas utilizadas. Por lo tanto, se ha mostrado únicamente una semilla en la imagen.

2.4.2. A280

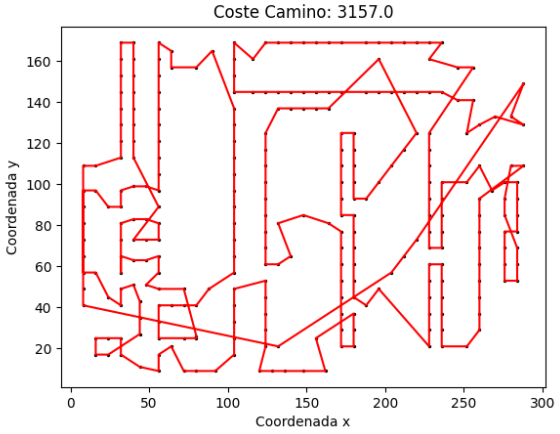


Figura 2: Resultados del algoritmo Greedy sobre el problema A280

En la imagen anterior se puede apreciar el camino generado por el algoritmo greedy implementado previamente para resolver el problema A280.

En este caso, no se muestra ninguna gráfica de evolución debido a que el algoritmo greedy únicamente genera un camino sin mejorarlo más adelante. Por lo tanto, no hay iteraciones o cambios en el camino durante la ejecución que requieran una representación gráfica.

Además, se ha presentado únicamente una solución, ya que el algoritmo greedy no tiene componentes aleatorios y siempre genera el mismo camino independientemente de la semilla utilizada.

3. Sistema de Hormigas

3.1. Introducción

En esta sección de la práctica, se ha implementado un algoritmo de sistema de hormigas. Se han utilizado un total de 30 hormigas para este propósito. En cuanto a la función de actualización de feromonas, se ha utilizado un factor α de 1 y un factor β de 2.

El objetivo de este algoritmo es simular una colonia de hormigas, donde cada individuo deja un rastro de feromonas a lo largo del camino que sigue. Con el tiempo, estas feromonas se evaporan gradualmente. Cada vez que una hormiga pasa por los nodos que conectan las aristas, contribuye a la cantidad de feromonas presente en cada arista.

Al final de la ejecución del algoritmo, se devuelve el mejor camino que se ha generado. Sin embargo, debido a restricciones de tiempo, el algoritmo solo pudo realizar 100 iteraciones dentro de los 3 minutos establecidos en el enunciado de la práctica. Por lo tanto, se ha agregado una condición de parada adicional que garantiza que se realicen al menos 400 iteraciones en total. Esta modificación busca mejorar la calidad de los resultados obtenidos por el algoritmo.

En resumen, en esta sección se ha implementado un algoritmo de sistema de hormigas, donde se han utilizado 30 hormigas y factores de actualización de feromonas de $\alpha = 1$ y $\beta = 2$. El algoritmo simula una colonia de hormigas que dejan rastros de feromonas en el camino, y devuelve el mejor camino generado. Además, se ha agregado una condición de parada para garantizar un mínimo de 400 iteraciones y mejorar la calidad de los resultados.

3.2. Análisis del algoritmo

El código proporcionado implementa un algoritmo basado en el Sistema de Hormigas para resolver el problema del viajante de comercio (TSP). En términos de coste computacional, el algoritmo presenta una complejidad dominada por el bucle principal y la actualización de las feromonas. El bucle principal se ejecuta hasta que se cumple una condición de tiempo de parada, mientras que la actualización de las feromonas implica cálculos en una matriz delta de tamaño num_arcos x num_arcos.

En cuanto a la capacidad de exploración y explotación, el algoritmo combina ambas estrategias utilizando feromonas y selección basada en probabilidades. Las feromonas guían a las hormigas hacia las regiones prometedoras del espacio de búsqueda (explotación), mientras que la selección de nodos durante la construcción de soluciones se basa en probabilidades calculadas a partir de las feromonas y las distancias entre los nodos (exploración). El equilibrio entre la exploración y la explotación está controlado por los parámetros alpha y beta, que determinan la influencia relativa de las feromonas y las distancias en la selección de nodos.

En términos de diversificación e intensificación, el algoritmo busca encontrar una solución óptima mediante la actualización de las feromonas (intensificación). Sin embargo, durante la fase de construcción de soluciones, existen oportunidades de diversificación debido al uso de una matriz delta para almacenar las actualizaciones de feromonas de cada hormiga, así como la selección basada en probabilidades que permite a las hormigas explorar diferentes caminos. En resumen, el algoritmo busca combinar tanto la diversificación como la intensificación para encontrar soluciones de alta calidad.

En resumen, el algoritmo implementado utiliza el Sistema de Hormigas para resolver el problema TSP. Su coste computacional está dominado por el bucle principal y la actualización de las feromonas. Combina estrategias de exploración y explotación a través del uso de feromonas y selección basada en probabilidades. Además, busca un equilibrio entre la diversificación y la intensificación para encontrar soluciones óptimas.

3.3. Código

3.3.1. Actualización de Feromonas

```
def actualizarFeromonas(distancias, soluciones, feromonas):
    num_hormigas = len(soluciones)  # Número de hormigas en la población
    num_arcos = len(soluciones[0])  # Número de arcos en una solución
    delta = np.zeros((num_arcos, num_arcos))  # Matriz para almacenar la cantidad de feromonas a actualizar
    costes_hormigas = np.array([Utils.funcionCoste(distancias, hormiga) for hormiga in soluciones])
    for hormiga in range(num_hormigas): # Actualizar las feromonas para cada hormiga
        # Obtener los arcos existentes en la solución de la hormiga
        arcos_existentes = existeArco(soluciones[hormiga], num_arcos)
        # Calcular el valor auxiliar para la actualización de feromonas
        aux = Utils.FACTOR_COSTE / costes_hormigas[hormiga]
        # Actualizar delta sumando el valor auxiliar en los arcos existentes de la hormiga
        delta[arcos_existentes] += aux
    # Aplicar evaporación y actualizar las feromonas
    feromonas *= (1 - Utils.EVAPORACION)
    feromonas += delta
```

3.3.2. Función de Transición

```
def transicion(feromonas, distancias, solucionHormiga):
    indices_validos = np.where(solucionHormiga != -1)[0]
    ultimoNodoVisitado = solucionHormiga[indices_validos[-1]]
    r = ultimoNodoVisitado
    # Obtiene la diferencia entre todos los nodos y los que se encuentran en solucionhormiga
    u_not_in_solucionHormiga = np.setdiff1d(np.arange(len(distancias)), solucionHormiga)
    feromonas_r = feromonas[r]
    # Realiza 1/distancia de forma más eficiente
    distancias_r_inv = np.reciprocal(distancias[r])
    sumaTotal = np.sum((feromonas_r[u_not_in_solucionHormiga] ** Utils.alpha) * (
                distancias_r_inv[u_not_in_solucionHormiga] ** Utils.beta))
    if sumaTotal <= 0: sumaTotal = 1e-10
    probabilidades = np.zeros(len(distancias))
    probabilidades[u_not_in_solucionHormiga] = (feromonas_r[u_not_in_solucionHormiga] ** Utils.alpha) * (
                distancias_r_inv[u_not_in_solucionHormiga] ** Utils.beta) / sumaTotal
    return probabilidades
```

3.3.3. Sistema de Hormigas

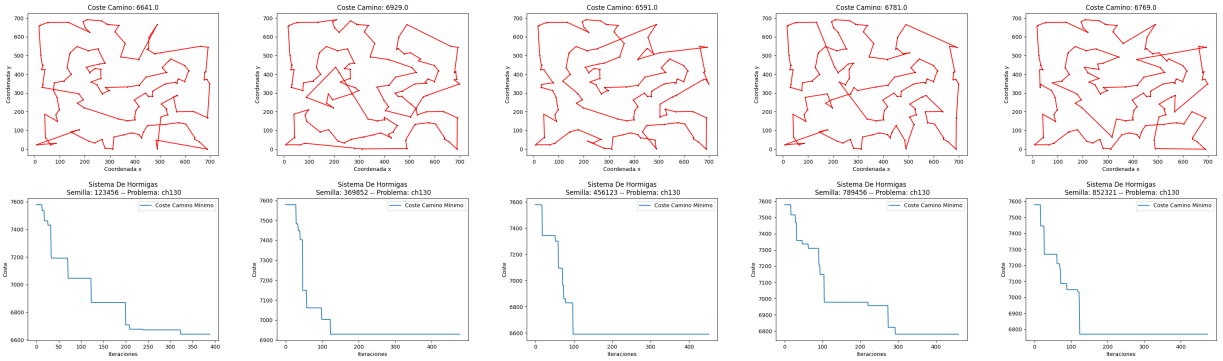
```
def SistemaHormigas(semilla, problema):
    np.random.seed(semilla);    random.seed(semilla)
    # Lectura de las coordenadas del problema:
    dimension, ciudades = Utils.leerFicheroTSP(f"..\\FicherosTSP\\{problema}.tsp")
    dimension = int(dimension)
    pathGreedy = greedy.tsp_greedy(semilla, problema)[1]
    # Inicializar matrices
    distancias = Utils.inicializarMatrizDistancias(ciudades)
    coste_greedy = Utils.funcionCoste(distancias, pathGreedy)
    feromonas = np.ones((dimension, dimension)) * (1 / (dimension * coste_greedy))
    coste = np.ones(Utils.numeroHormigas) * float('inf')
```

```
    mejorActual = coste_greedy;    mejorGlobal = coste_greedy
    mejorHormiga = pathGreedy;    mejorHormigaActual = pathGreedy
    # Comienza el algoritmo
    start = time.time()
    while (time.time() - start) < Utils.TiempoParada(problema):
        solucionHormigas = np.ones((Utils.numeroHormigas, dimension), dtype=int) * -1
        for hormiga in range(len(solucionHormigas)):  # Para cada hormiga
            solucionHormigas[hormiga][0] = 0  # El primer nodo es el 0
            for nodo in range(1, dimension):  # Para cada nodo
                probabilidades = transicion(feromonas, distancias, solucionHormigas[hormiga])
                # Elegir un nodo en funcion de las probabilidades calculadas
                solucionHormigas[hormiga][nodo] = elegirNodo(probabilidades)
            coste[hormiga] = Utils.funcionCoste(distancias, solucionHormigas[hormiga])
            if coste[hormiga] < mejorActual:
                mejorActual = coste[hormiga]
                mejorHormigaActual = solucionHormigas[hormiga].copy()
        actualizarFeromonas(distancias, solucionHormigas, feromonas)
        numeroEvaluaciones += Utils.numeroHormigas
        if mejorActual < mejorGlobal:
            mejorGlobal = mejorActual
            mejorHormiga = mejorHormigaActual.copy()
    return mejorGlobal, mejorHormiga, numeroEvaluaciones, MejorCosteIteracion
```

3.4. Resultados

A continuación se mostrarán los resultados obtenidos para ambos problemas propuestos en el enunciado. Tanto el camino mínimo encontrado como la evolución de la longitud del camino mínimo obtenido por el algoritmo en cada iteración.

3.4.1. CH130



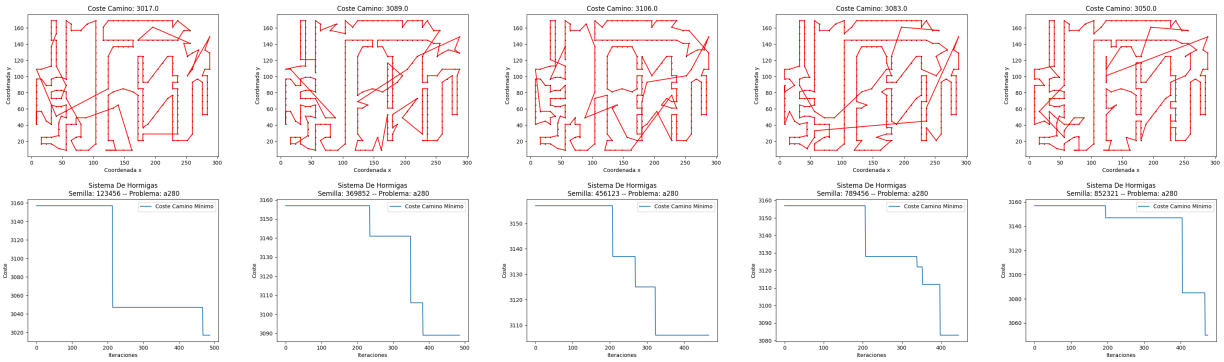
Cuadro 2: Resultados obtenidos con el algoritmo de Sistema de Hormigas para el problema ch130.

En la tabla 2 podemos ver en la fila superior el camino generado por la mejor hormiga en cada una de las semillas que se ha utilizado para ejecutar el algoritmo junto con el coste de dicho camino generado. Por otro lado, en la fila inferior podemos ver la evolución del coste del mejor camino durante toda la ejecución del algoritmo de sistema de hormigas.

Con este algoritmo hemos conseguido disminuir el coste que obtuvimos con el algoritmo greedy en la sección anterior (7579). Al ser un algoritmo heurístico no conseguimos llegar al coste mínimo global del problema. Sin embargo, en la gráfica de fitness podemos observar que el camino mínimo converge, en la

mayoría de ejecuciones, antes de la mitad del límite de tiempo.

3.4.2. A280



Cuadro 3: Resultados obtenidos con el algoritmo de Sistema de Hormigas para el problema a280.

En la tabla 3 podemos observar, al igual que en el problema anterior, los caminos generados por la mejor hormiga al final de la ejecución del algoritmo sobre las cinco semillas y, en la fila inferior, la evolución del coste del camino mínimo durante toda la ejecución del algoritmo.

Cabe destacar que, a diferencia del problema anterior, aquí no consigue mejorar demasiadas veces el algoritmo de sistema de hormigas. Puede ser por falta de tiempo o debido al tamaño considerablemente superior del problema.

De todas formas, el algoritmo consigue mejorar al algoritmo greedy al igual que en el problema anterior pasando de 3157 a una media de 3069.

4. Sistema de Hormigas Elitista

4.1. Introducción

En esta sección de la práctica, se realizará una modificación al algoritmo previo de *sistema de hormigas* para convertirlo en un sistema de hormigas elitista. En esta nueva aproximación, se cambiará el número de hormigas utilizadas en la ejecución del algoritmo a 15. Los demás parámetros se mantendrán iguales a los utilizados en el algoritmo anterior.

La idea detrás de este enfoque elitista es fomentar la exploración del espacio de soluciones al permitir que un número reducido de hormigas, en este caso 15, realicen un seguimiento más intensivo del mejor camino encontrado hasta el momento. La mejor hormiga obtenida hasta el momento depositará aportará más feromonas en el camino que ha escogido. Esto aumentará la probabilidad de que las otras hormigas sigan dicha ruta.

En resumen, en esta sección se modificará el algoritmo previo de sistema de hormigas para convertirlo en un sistema de hormigas elitista. Se utilizarán 15 hormigas en lugar de las 30 anteriores, manteniendo los mismos parámetros de actualización de feromonas. Esta modificación tiene como objetivo promover la exploración de soluciones óptimas al permitir que el mejor camino aporte más feromonas a la colonia.

4.2. Análisis del algoritmo

En esta sección se realizará un análisis detallado del código implementado para el algoritmo de Sistema de Hormigas Elitista (SHE) utilizado en la resolución del problema del viajante de comercio (TSP). A continuación, se describen los aspectos clave relacionados con su rendimiento y comportamiento.

El algoritmo implementado presenta un coste computacional que depende de varios factores. El bucle principal del algoritmo itera hasta que se cumple una condición de tiempo de parada establecida. Dentro de este bucle, se realizan operaciones como la construcción de soluciones por parte de las hormigas, el cálculo de costes y la actualización de las feromonas. Además, se utiliza la función *Utils.funcionCoste* para calcular el coste de una solución. En general, el tiempo de ejecución del algoritmo es proporcional al número de hormigas, dimensiones del problema y al tiempo de parada establecido.

El algoritmo de Sistema de Hormigas Elitista combina estrategias de exploración y explotación para buscar soluciones de alta calidad. La exploración se lleva a cabo mediante la selección de nodos basada en probabilidades. Cada hormiga elige su siguiente nodo en función de las feromonas depositadas y las distancias entre los nodos. Esto permite que las hormigas exploren diferentes caminos y eviten caer en una solución local óptima. Por otro lado, la explotación se logra utilizando la información proporcionada por la mejor hormiga hasta el momento (mejorHormiga) y la actualización de las feromonas en función de su coste. Esta combinación de estrategias permite encontrar soluciones óptimas al problema del TSP.

Durante la ejecución del algoritmo, se busca tanto la diversificación como la intensificación. La diversificación se logra debido a que cada hormiga construye su propia solución utilizando la selección de nodos basada en probabilidades. Esto permite que las hormigas exploren diferentes caminos y eviten converger prematuramente hacia una solución subóptima. Por otro lado, la intensificación se lleva a cabo mediante la actualización de las feromonas. Las hormigas depositan feromonas en los arcos que forman parte de la mejor solución actual (mejorHormiga), lo que aumenta la probabilidad de que las hormigas futuras sigan ese camino y refuerza la búsqueda en áreas prometedoras del espacio de búsqueda.

4.3. Código

En este algoritmo lo único que cambia es la función de actualización de feromonas, en la que se añade un nuevo término que representa el aporte del mejor camino hasta ese punto de la ejecución.

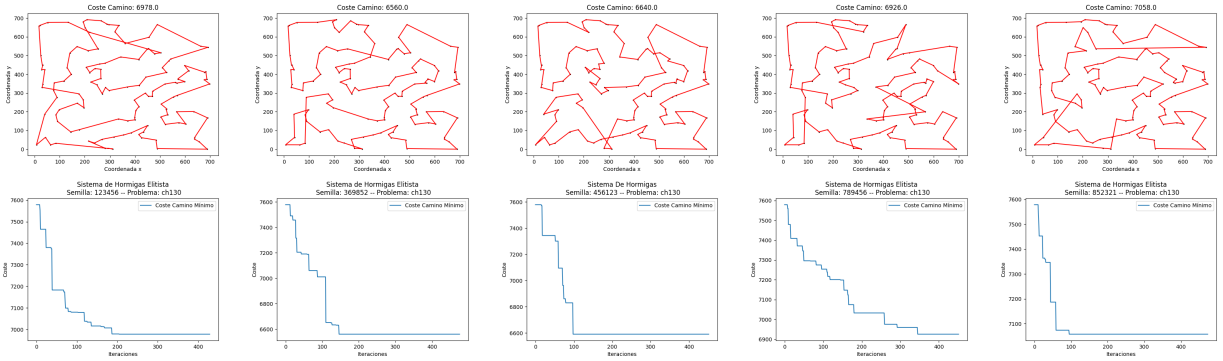
4.3.1. Actualización de Feromonas

```python
def actualizarFeromonas(distancias, soluciones, feromonas, mejorHormiga):
    num_hormigas = len(soluciones)  # Número de hormigas en la población
    num_arcos = len(soluciones[0])  # Número de arcos en una solución
    delta = np.zeros((num_arcos, num_arcos))  # Matriz para almacenar la cantidad de feromonas a actualizar
    # Calcular los costes de todas las hormigas en paralelo utilizando operaciones vectorizadas
    costes_hormigas = np.array([Utils.funcionCoste(distancias, hormiga) for hormiga in soluciones])
    # Calcular el coste de la mejorHormiga
    mejor_coste_hormiga = Utils.funcionCoste(distancias, mejorHormiga)
    # Actualizar las feromonas para cada hormiga
    for hormiga in range(num_hormigas):
        # Obtener los arcos existentes en la solución de la hormiga
        arcos_existentes = existeArco(soluciones[hormiga], num_arcos)

        # Calcular el valor auxiliar para la actualización de feromonas
        aux = Utils.FACTOR_COSTE / costes_hormigas[hormiga]

        # Actualizar delta sumando el valor auxiliar en los arcos existentes de la hormiga
        delta[arcos_existentes] += aux

    mejor_coste_hormiga = 1e-10 if mejor_coste_hormiga == 0 else mejor_coste_hormiga
    # Actualizar delta sumando el término correspondiente a la mejorHormiga
    delta[existeArco(mejorHormiga, num_arcos)] += Utils.numeroHormigasElitista / mejor_coste_hormiga

    # Aplicar evaporación y actualizar las feromonas
    feromonas *= (1 - Utils.EVAPORACION)
    feromonas += delta
```

4.4. Resultados

A continuación se mostrarán los resultados obtenidos para ambos problemas propuestos en el enunciado. Tanto el camino mínimo encontrado como la evolución de la longitud del camino mínimo obtenido por el algoritmo en cada iteración.

4.4.1. CH130

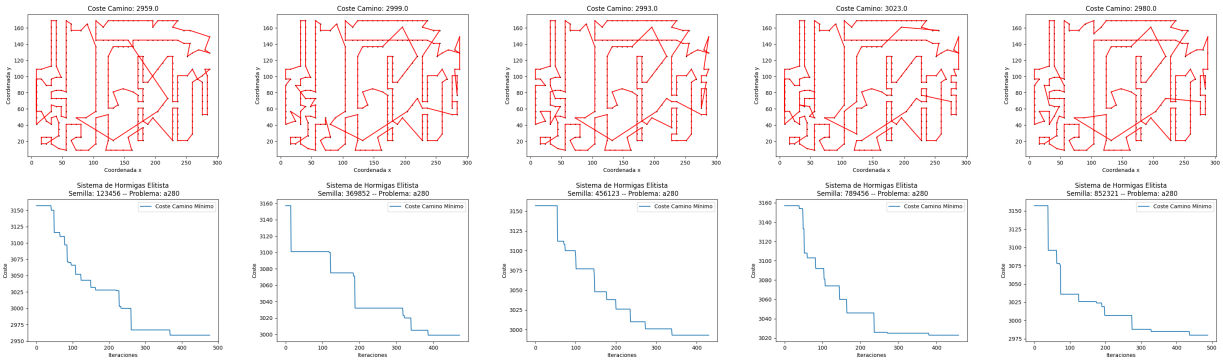


Cuadro 4: Resultados obtenidos con el algoritmo de Sistema de Hormigas Elitista para el problema ch130.

En la tabla 4 podemos observar, como con el algoritmo anterior, la fila superior donde se representa el mejor camino obtenido con la mejor hormiga de toda la ejecución del algoritmo de sistema elitista de hormigas para cada una de las semillas seleccionadas junto con el coste de cada uno. Por otro lado, en la fila inferior podemos ver la gráfica que representa el *fitness* del mejor camino en cada iteración del algoritmo para el problema de ch130.

Podemos observar que los resultados que se han obtenido han mejorado el coste que obteníamos con el algoritmo greedy. Sin emabargo, el coste medio no ha superado al algoritmo de sistema de hormigas normal. No obstante, podemos ver que las gráficas de *fitness* consiguen mejorar un número mayor de veces que con el anterior algoritmo debido al *refuerzo* en el camino de la mejor hormiga hasta el momento.

4.4.2. A280



Cuadro 5: Resultados obtenidos con el algoritmo de Sistema de Hormigas Elitista para el problema a280.

En este problema podemos ver que el coste del mejor camino generado por la mejor hormiga de cada una de las iteraciones sí que consigue mejorar el coste que tenían tanto en el algoritmo greedy como en el algoritmo de sistema de hormigas normal.

Además, como en el problema anterior, el número de veces que mejora el camino es muy superior al conseguido con el algoritmo de sistema de hormigas visto anteriormente. Esto provoca que, al reforzar al comienzo, el coste del camino *greedy*, el mejor camino sea más probable que mejore.

5. Resultados

5.1. CH130

A continuación se muestra la tabla comparativa de los resultados y llamadas a la función de evaluación obtenidas para el problema ch130.

	Problema CH130					
	Greedy		**Sistema de Hormigas**		**Sistema de Hormigas Elitista**	
	Coste	Ev.	Coste	Ev.	Coste	Ev.
Ejecución 1	7579	1	6641	28500	6978	25800
Ejecución 2	7579	1	6781	27660	6926	27120
Ejecución 3	7579	1	6591	27120	6640	28620
Ejecución 4	7579	1	6929	28620	6560	28560
Ejecución 5	7579	1	6769	28500	7058	28440
Media	7579.0	1.0	6742.2	28080	6832.4	27708.0
Desv. Típica	0.0	0.0	132.5	660.0	219.13	1232.85

Cuadro 6: Resultados para el problema CH130.

En la tabla anterior podemos observar como el algoritmo greedy consigue el peor resultado en todas las ejecuciones del algoritmo. Esto es algo normal, ya que lo que pretendemos con los algoritmos de hormigas es mejorar dicha solución.

En cuanto a los algoritmos basados en colonia de hormigas podemos decir, que para este primer problema, el que obtiene un mejor coste es el algoritmo de sistema de hormigas normal con un coste medio de 6742,2, mientras que el algoritmo de hormigas elitista obtiene un coste medio de 6823,3. Estos resultados pueden ser debidos a la sobrecarga de la CPU en el momento de ejecutar los algoritmos, ya que, al tener una condición de parada relativa al tiempo de ejecución, no tienen por qué realizar las mismas iteraciones en cada ejecución.

En cuanto al número de evaluaciones podemos decir que ambos algoritmos realizan un número similar de llamadas a la función de coste, sin embargo, el número del algoritmo de sistema de hormigas elitista es ligeramente inferior a su adversario. En contraparte, la robustez del algoritmo de sistema de hormigas elitista en cuanto al número de llamadas a la función de evaluación es mucho menor que la del algoritmo de hormigas normales.

En cualquier caso, ambos algoritmos cumplen la función de mejorar el coste del algoritmo greedy en casi 800 unidades.

5.2. A280

A continuación se muestra la tabla comparativa de los resultados obtenidos junto a las llamadas a la función de evaluación para el problema a280.

Cabe mencionar que, los nodos 171 y 172 eran iguales. Se ha optado por incrementar una unidad en la coordenada **y** del nodo **172**.

	Problema A280					
	Greedy		**Sistema de Hormigas**		**Sistema de Hormigas Elitista**	
	Coste	Ev.	Coste	Ev.	Coste	Ev.
Ejecución 1	3157	1	3017	29220	2959	28680
Ejecución 2	3157	1	3083	26760	3023	27540
Ejecución 3	3157	1	3106	28020	2993	25860
Ejecución 4	3157	1	3089	29160	2999	28320
Ejecución 5	3157	1	3050	28500	2980	29340
Media	3157.0	1.0	3069.0	28332	2990.8	27948
Desv. Típica	0.0	0.0	35.46	1008.82	23.65	1335.86

Cuadro 7: Resultados para el problema A280.

En la tabla anterior podemos observar como el algoritmo greedy, al igual que en el problema anterior, obtiene los peores resultados en todas las ejecuciones. Esto quiere decir que los algoritmos basados en colonias de hormigas han funcionado y han permitido mejorar la solución obtenida por el algoritmo voraz.

En cuanto a la comparación entre ambos algoritmos OCH podemos decir que, en este problema concreto, el algoritmo de sistema de hormigas elitista consigue mejores resultados en todas las ejecuciones obteniendo una media de 2990,8 a diferencia del sistema de hormigas normales que obtiene un resultado medio de 3069. Además, el algoritmo de SHE obtiene una desviación típica menor, haciéndolo más robusto en cuanto a los resultados obtenidos.

En cuanto al número de llamadas a la función de evaluación, el algoritmo SHE también obtiene los mejores resultados, pero con una robustez un poco menor que el algoritmo SH.

En este problema hemos visto que el algoritmo de sistema de hormigas elitista consigue unos mejores resultados que sus adversarios convirtiéndolo en la mejor opción para ejecutar este problema.

6. Conclusiones

En el transcurso de esta práctica, hemos explorado e implementado dos algoritmos de optimización basados en colonias de hormigas (OCH). Además, hemos realizado una comparación de su eficacia junto con un algoritmo greedy para resolver instancias del problema del viajante de comercio, específicamente las denominadas **ch130** y **a280**.

Nuestros resultados revelan que la elección del algoritmo depende en gran medida del problema en cuestión. En el caso de **ch130**, hemos observado que el algoritmo de sistema de hormigas normales obtiene mejores resultados en comparación con los demás. Sin embargo, en el problema **a280**, el sistema de hormigas elitista ha demostrado obtener los mejores resultados.

Es importante destacar que estos algoritmos resultan especialmente útiles en situaciones en las que el tiempo de espera no es una restricción crucial. La experimentación y ejecución de estos algoritmos demandan un tiempo considerable para lograr una optimización óptima.