INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE COMPUTO

2024

SESIÓN DE LABORATORIO 7: ALGORITMO DE OPTIMIZACIÓN DE COLONIA DE HORMIGAS

ALGORITMOS BIOINSPIRADOS

Elaborado por: Flores Estopier Rodrigo

Boleta: 2021630260

Profesor: Rosas Trigueros Jorge Luis

Grupo: 6CV2

Fecha de realización: 31/10/2024

Fecha de entrega: /10/2024

Semestre: 25-1

Sesión de Laboratorio 7: Algoritmo de optimización de colonia de hormigas

Contenido

1. Marco Teórico

El Algoritmo de Optimización por Colonia de Hormigas (ACO) es una técnica bioinspirada que emula el comportamiento colectivo de las hormigas en la naturaleza, específicamente en su búsqueda de alimentos. Cuando una hormiga encuentra comida, deja un rastro de feromonas en el camino, lo que aumenta la probabilidad de que otras hormigas sigan ese mismo camino. Este proceso permite que las hormigas optimicen las rutas hacia las fuentes de alimento. Aplicado a problemas de optimización, como el Problema del Viajero (TSP), ACO busca encontrar la ruta más corta que permita visitar un conjunto de ciudades una sola vez y regresar a la ciudad inicial.

Componentes del Algoritmo ACO:

1. **Feromonas**: Los rastros de feromonas guían a las hormigas hacia soluciones prometedoras. Cuanto más corto es un camino, más feromonas se depositan, favoreciendo rutas eficientes.
2. **Heurística**: Se basa en la distancia entre ciudades, y es inversamente proporcional a la longitud del camino. Las hormigas prefieren caminos cortos, ponderados por este valor heurístico.
3. **Evaporación de Feromonas**: Evita que el algoritmo se estanque en soluciones subóptimas, al reducir la intensidad de las feromonas a lo largo del tiempo.
4. **Construcción de Rutas**: Cada hormiga construye una ruta seleccionando ciudades según la influencia combinada de las feromonas y la heurística. Las ciudades no visitadas se eligen probabilísticamente, normalizando las probabilidades para garantizar una elección aleatoria pero informada.

Aplicación al Problema del Viajero (TSP):

El TSP es un clásico problema de optimización combinatoria, donde se busca minimizar la distancia total al visitar un conjunto de ciudades y volver al punto de origen. El ACO es eficaz para este problema gracias a su capacidad para explorar múltiples soluciones simultáneamente y ajustar las rutas basándose en el éxito de iteraciones anteriores.

2. DESARROLLO

CODIFICACIÓN

Se utilizaron las siguientes librerías para la realización del algoritmo.

```
import numpy as np
import random
```

- numpy: Se utilizo para utilizar elementos matemáticos como el infinito, el cuas se establece como valor principal de la distancia menor global.
- random: Nos permitirá construir las rutas de las hormigas, haciendo que estas elijan una ciudad de forma “aleatoria”, donde además se toma en cuenta su valor de probabilidad.

Posteriormente, se construye la matriz de distancias que nos indica que distancia hay entre las ciudades.

```
# La primera columna ni fila se utilizan
M = 2 * np.ones([11, 11])
M[1][3] = 1; M[3][5] = 1; M[5][7] = 1; M[7][9] = 1
M[9][2] = 1; M[2][4] = 1; M[4][6] = 1; M[6][8] = 1
M[8][10] = 1

print(M)
```

Parámetros

Se consideraron los siguientes parámetros para la realización del algoritmo ACO:

- Numero de hormigas por iteración: 10
- Valor Alpha: 1.0 nos indica que tanto influyen las feromonas en la decisión de una hormiga al tomar un camino.
- Valor beta: 2.0 nos indica que tanto influye el valor de la heurística en un camino para la decisión de una hormiga.
- Tasa de evaporación: 0.3 nos indica que tanto se evaporan las feromonas entre cada iteración

- Depósito de feromonas: Nos indica el valor base de depósito de feromonas de una hormiga.

```
# Parámetros ACO
num_ants = 10
#num_iterations = 100
alpha = 1.0          # Influencia de las feromonas
beta = 2.0           # Influencia de la heurística (inverso de la
distancia)
evaporation_rate = 0.3
pheromone_deposit = 1.0
```

Matrices

Se utilizarán 3 matrices para la implementación del algoritmo.

1. La matriz de distancias mencionada anteriormente.
2. La matriz de feromonas, la cual inicializamos en unos en todas sus posiciones.
3. La matriz de heurística, la cual contiene el inverso de la distancia entre dos ciudades. Un viaje de una ciudad a otra tendrá mayor valor cuando tenga una menor distancia.

```
# Inicializamos la matriz de feromonas
pheromone_matrix = np.ones_like(M)

# Inicializamos la matriz de heurística (inverso de la distancia)
heuristic_matrix = 1 / (M)
```

Funcion de aptitud

En este caso, se consideró a la funcion de aptitud o fitness, el cálculo de la distancia total de una ruta. La funcion recibe un arreglo que contiene la secuencia de la ruta y calcula con ayuda de la matriz de distancias, cual fue la distancia total recorrida por la hormiga.

```
# Función de aptitud (ditancia total de la ruta)
def route_distance(route):
    distance = 0
    for i in range(len(route) - 1):
        distance += M[route[i]][route[i + 1]]
    distance += M[route[-1]][route[0]]  # Regresar al punto inicial
    return distance
```

Funcion de construcción de ruta

Esta es una de las funciones principales del algoritmo. A continuación, se explicará de forma seccionada cómo funciona este método.

La funcion recibe una ciudad inicial donde comenzara su recorrido la hormiga. Esta ciudad se añade a la lista de ciudades visitadas.

```
def construct_route(start_city):
    route = [start_city]
    visited = set(route)
```

Posteriormente, para cada una de las ciudades restantes se realiza lo siguiente:

```
for _ in range(9):  # 10 ciudades en total , ya se visitó la inicial
```

Obtenemos la ciudad actual donde esta la hormiga.

```
        current_city = route[-1]
        probabilities = []
```

Realizamos el cálculo de probabilidad para cada ciudad que no ha sido visitada, considerando que se viaja desde la ciudad actual. Aquí consideramos los parámetros Alpha y beta , donde multiplicamos el valor de feromonas en la ruta desde la ciudad actual hasta la ciudad objetivo por el valor de heurística en la misma ruta, cada uno de los valores ponderados por Alpha y beta respectivamente.

```
        # Calcular probabilidades para las ciudades no visitadas
        for next_city in range(1, 11):
            if next_city not in visited:
                prob = (pheromone_matrix[current_city][next_city] **
alpha) * \
                       (heuristic_matrix[current_city][next_city] ** beta)
                probabilities.append((next_city, prob))
```

Antes de elegir una ciudad con base en las probabilidades, debemos normalizar la probabilidad, dividiendo cada probabilidad por la suma de todas las probabilidades.

```
        # Normalizar y seleccionar aleatoriamente la ciudad siguiente
        total_prob = sum(prob for _, prob in probabilities)
```

```
        probabilities = [(city, prob / total_prob) for city, prob in
probabilities]
```

Por último, en este apartado realizamos la elección aleatoria de una ciudad considerando sus probabilidades.

La hormiga elige una ciudad, esta se añade a la ruta y a la lista de ciudades visitadas. Así se va construyendo la ruta de una sola hormiga.

```
        next_city = random.choices([city for city, _ in probabilities],
                                   weights=[prob for _, prob in
probabilities])[0]

        route.append(next_city)
        visited.add(next_city)
```

Funcion para actualizar feromonas

En este caso, no se considero el calculo de feromonas de manera completa. Ya que solo realizamos el deposito de feromonas en la mejor ruta hasta el momento, y no en cada ruta y cantidad de hormigas que hayan pasado por esa ruta.

Esta funcion recibe la mejor ruta hasta ahora en forma de vector, posteriormente se realiza la evaporación de todas las celdas de la matriz.

Por último, se realiza el deposito de feromonas en las celdas pertenecientes a la mejor ruta, asegurándonos que se realiza el depósito en ambos sentidos.

```
# Funcion para actualizar las feromonas
def update_pheromones(best_route):
    global pheromone_matrix
    pheromone_matrix *= (1 - evaporation_rate)  # Evaporación de toda la
matriz

    # Depósito de feromonas solo en la mejor ruta
    for i in range(len(best_route) - 1):
        city_a = best_route[i]
        city_b = best_route[i + 1]
        pheromone_matrix[city_a][city_b] += pheromone_deposit /
route_distance(best_route)
        pheromone_matrix[city_b][city_a] += pheromone_deposit /
route_distance(best_route)
```

Algoritmo ACO

Se utilizan las funciones explicadas anteriormente para llevar a cabo la ejecución del algoritmo de colonia de hormigas.

Esta funcion recibe el número de iteraciones que se quieren ejecutar del algoritmo. Además, utiliza las variables de mejor ruta y mejor distancia.

```
def ACO(num_iterations):
    global best_route, best_distance,count
```

En un ciclo for realizamos las iteraciones establecidas.

```
    for iteration in range(num_iterations):

        # Cada hormiga construye una ruta, con una ciudad de inicio
aleatoria
```

En otro ciclo for anidado se realizan los recorridos de cada hormiga.

```
        for ant in range(num_ants):
```

Por cada hormiga se selecciona una ciudad al azar para comenzar su recorrido. Se construye su recorrido y se obtiene la distancia de dicho recorrido.

```
            start_city = random.randint(1, 10)
            route = construct_route(start_city)
            route_dist = route_distance(route)
```

Verificamos si el recorrido de la hormia tiene mejor valor de aptitud que el mejor actual.

```
            # Actualizamos la mejor ruta
            if route_dist < best_distance:
                best_distance = route_dist
                best_route = route
```

Una vez que se realizan los recorridos de todas las hormigas, se actualizan los valores de feromonas para continuar con la siguiente iteración.

```
        # Actualizar feromonas en la mejor ruta
        update_pheromones(best_route)

        count += 1
        print(f" {count}: Best distance = {best_distance}, Best route =
{best_route}")
```

Por último, inicializamos los valores de mejor ruta y distancia para que sean utilizados en el algoritmo, y llamamos a la funcion del algoritmo para comenzar la ejecución.

```
#Definimos las variables para guardar la mejor ruta y distancia
best_route = None
best_distance = np.inf
count = 0
# Ejecutar el algoritmo
ACO(5)
```

EJECUCIÓN

```
[[2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 1. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 1. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 1. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 1. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 1. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2. 1. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2. 2. 1.]
 [2. 2. 1. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]]
 1: Best distance = 15.0, Best route = [7, 9, 4, 2, 1, 3, 5, 6, 8, 10]
 2: Best distance = 15.0, Best route = [7, 9, 4, 2, 1, 3, 5, 6, 8, 10]
 3: Best distance = 15.0, Best route = [7, 9, 4, 2, 1, 3, 5, 6, 8, 10]
 4: Best distance = 14.0, Best route = [5, 7, 9, 2, 4, 3, 6, 8, 10, 1]
 5: Best distance = 14.0, Best route = [5, 7, 9, 2, 4, 3, 6, 8, 10, 1]
```

Ilustración 1. Ejecución del algoritmo hasta la iteración 5.

```
ACO(5)

 11: Best distance = 11.0, Best route = [4, 6, 8, 10, 1, 3, 5, 7, 9, 2]
 12: Best distance = 11.0, Best route = [4, 6, 8, 10, 1, 3, 5, 7, 9, 2]
 13: Best distance = 11.0, Best route = [4, 6, 8, 10, 1, 3, 5, 7, 9, 2]
 14: Best distance = 11.0, Best route = [4, 6, 8, 10, 1, 3, 5, 7, 9, 2]
 15: Best distance = 11.0, Best route = [4, 6, 8, 10, 1, 3, 5, 7, 9, 2]
```

Ilustración 2. Ejecución del algoritmo hasta la iteración 15.

3. CONCLUSIONES

La implementación del algoritmo para resolver el TSP mostró que es un método útil para abordar problemas de optimización combinatoria, siendo que no es requerido tener conocimientos avanzados en matemática y probabilidad. Se observó que la elección adecuada de los parámetros (como la tasa de evaporación y los valores de alpha y beta) es fundamental para un buen funcionamiento. Un ajuste erróneo puede resultar en la convergencia prematura que deje al algoritmo estancado en una solución local o en una exploración ineficiente del espacio de soluciones.

Una observación relevante fue la manera en que las feromonas guían a las hormigas hacia soluciones prometedoras, pero también cómo la evaporación evita que el algoritmo se estanque en mínimos locales. Un experimento personal realizado fue ajustar la tasa de evaporación y observar cómo afecta la diversidad de las rutas generadas; un valor más alto permitió una exploración más exhaustiva.

Para un mejor desarrollo, se podría experimentar con distintos tamaños de colonia y más iteraciones para analizar cómo estos cambios influyen en la calidad de las soluciones. Visualizar las rutas y el progreso de las hormigas a lo largo de las iteraciones también sería una mejora para comprender mejor el comportamiento del ACO.

4. BIBLIOGRAFÍA

[1] "Ant colony optimization", *Scholarpedia.org*. [En línea]. Disponible en: http://www.scholarpedia.org/article/Ant_colony_optimization. [Consultado: 01-nov-2024].

[2] "Introduction to ant colony optimization", *GeeksforGeeks*, 15-may-2020. [En línea]. Disponible en: https://www.geeksforgeeks.org/introduction-to-ant-colony-optimization/. [Consultado: 01-nov-2024].

[3] M. Kuo, "Algorithms for the travelling salesman problem", *Routific.com*, 02-ene-2020. .