Inteligencia Artificial

Ricardo Leguizamon[1,1], Diego Seo[1], Lorenzo Cabrera[1]
1 ingeniería en Informática, Facultad Politécnica de la Universidad Nacional de Asunción, San Lorenzo, Paraguay

Resumen. Este trabajo tiene como objetivo presentar un ensayo referente al problema del agente viajero (TSP por sus siglas en inglés) donde su representación ha resuelto varios problemas que pueden ser moldeados con base en las características del algoritmo base de TSP o de sus múltiples variables.

Se presenta en la sección de desarrollo el algoritmo base y una descripción de TSP con base en las características que han propuesto diferentes autores, y la aplicación de TSP como simulación a problemas reales . Por último, en la conclusión se aborda el tema de TSP como un paradigma que se puede emplear en situaciones donde se involucran puntos de control y costo entre los nodos.

Palabras Clave: Problema del Agente Viajero, Inteligencia Artificial

Keywords: Travel Salesman Problem (TSP)

1 Introducción

Este trabajo tiene como objetivo mostrar un marco comparativo entre los Algoritmos MOACO tales como MOACS y los Algoritmos MOEA tales como NSGA. El Marco Comparativo tiene como límite una evaluación hecha con 2 instancias de prueba para cada tipo de problema para TSP (Traveling Salesman Problem), tales problemas se definen como problemas del tipo NP-Difícil.

La primera solución reportada para resolver el problema del Agente Viajero fue en 1954, para resolver una instancia de 49 ciudades donde un agente viajero desea visitar un conjunto de ciudades, asignándoles un costo por visitar ciudades contiguas (distancia de traslado entre dos ciudades). Para esta solución se propusieron 2 condiciones: regresar a la misma ciudad de la cual partió y no repetir ciudades con el objetivo de encontrar una ruta o un camino con el menor costo posible.

2 Formulación del problema

Se tiene un número de nodos (ciudades, localidades, tiendas, empresas, etc.) que deben ser visitados por una entidad (persona, agente viajero, automotor, avión, autobús, etc.), sin visitar 2 veces el mismo nodo. Si tenemos 3 nodos (a, b y c) por visitar, entonces tendríamos una función de combinaciones sin repetición c(3,2), es decir, tendríamos 6 posibles soluciones: abc, acb, bac, bca, cab, cba, para el caso de 4 nodos tendríamos 12 combinaciones, para 10 nodos tendríamos 90 combinaciones, para 100 ciudades tendríamos 9,900 combinaciones y así sucesivamente.

Algoritmo de MOEA (Multiobjective Optimization Evolutionary Algorithm)

Son herramientas algorítmicas desarrolladas recientemente para la resolución de MOPs.

Características:
a) resultan promisorios para la integración de los aspectos de búsqueda y de toma de decisiones que plantean los MOPs;
b) pueden realizar búsquedas aún en espacios ilimitados y altamente complejos;
c) tienen la habilidad de mantener toda una población de soluciones;
d) existen pocas alternativas –aparte de ellos– para el tratamiento adecuado de MOPs.

La característica principal de un MOEA es que optimiza simultáneamente un conjunto múltiple de objetivos.

Non-dominated Sorting Genetic Algorithm o NSGA

Es un algoritmo genético basado en no dominancia, La idea detrás de NSGA es la utilización de un método de selección basado en rangos que se asignan según las relaciones de dominancia entre los individuos. De esta manera, se da prioridad en la selección a los individuos dominantes, mientras que los dominados tendrán menores posibilidades de tener descendencia., entre sus principales características se pueden citar:
a) Convergencia rápida de la población hacia las regiones no dominadas.
b) El procedimiento para compartir fitness ayuda a distribuir las soluciones encontradas a lo largo de todo el ámbito de búsqueda.
c) No tiene límites en cuanto a la cantidad de funciones objetivo a optimizar.
d) Puede lidiar tanto con problemas de maximización como con problemas de minimización, sencillamente cambiando la forma de determinar la dominancia entre los individuos.

Algoritmo de Moaco

Moaco utiliza más el enfoque de Colonia de Hormigas con este algoritmo en donde este algoritmo MOACS que viene de la sigla en inglés Multi-Objective Ant Colony System implementado para la solución del Problema del Vendedor Viajante con dos objetivos utiliza este algoritmo en busca de minimizar los dos objetivos que se encuentran en el conjunto de datos y que pueden encontrarse en un frente con los caminos con sus costos minimizados.

```
    import os
    import pandas as pd
    import numpy as np
    import math
    from nsga import lectura_archivo, Individuo
    import matplotlib.pyplot as plt
    # Inicializar parametros
    """
    m = numero de hormigas
    q0 = probabilidad de exploracion o explotacion
    """
    max_iter = 20
    beta = 0.10
    m = 20
    ciudades = []
    q0 = 0.6
    tau0 = 0.1*10**-20
    rho = 0.02
    n = 0
    class Colonia:
        def __init__(self, tam=m):
            """Clase que representa la colonia de hormigas. tam es el numero de hormigas.
            Las hormigas son inicializadas aleatoriamente."""
            self.tam = tam
            self.miembros = [Hormiga(i+1) for i in range(tam)]
            self.pareto_set = self.miembros
            for miembro in self.miembros:
                miembro.construir_camino_sin_feromonas()
            self.actualizar_frente_pareto()
            self.actualizacion_global_feromonas()

        def encontrar_no_dominados(self, hormigas):
            """Encontrar las hormigas no dominadas en la iteracion actual."""
            pareto_set = []
            for hormiga1 in hormigas:
                hormiga1_dominada = False
                for hormiga2 in hormigas:
                    if not hormiga1_dominada and hormiga2 != hormiga1:
                        if hormiga1.costos_camino[0] >= hormiga2.costos_camino[0] and hormiga1.costos_camino[1] >=
hormiga2.costos_camino[1] and (hormiga1.costos_camino[0] > hormiga2.costos_camino[0] or hormiga1.costos_camino[1] >
hormiga2.costos_camino[1]):
                            hormiga1_dominada = True
                if not hormiga1_dominada:
                    pareto_set.append(hormiga1)
            if len(pareto_set) > 0:
                return pareto_set

        def actualizar_frente_pareto(self):
            no_dominados = self.encontrar_no_dominados(self.miembros)
            hormigas = set(no_dominados).union(set(self.pareto_set))
            self.pareto_set = copy.deepcopy(self.encontrar_no_dominados(list(hormigas)))

        def nuevos_caminos(self):
            """Envia a todas las hormigas de la colonia en busca de un nuevo camino."""
            for miembro in self.miembros:
                miembro.construir_nuevo_camino()

        def actualizacion_global_feromonas(self):
            """Actualizacion global de feromonas. Debe ser llamado despues de que se haya encontrado a la mejor
hormiga."""

            hormigas_optimas = len(self.pareto_set)
            promedio_costo_1 = sum([hormiga.costos_camino[0] for hormiga in self.pareto_set]) / hormigas_optimas
            promedio_costo_2 = sum([hormiga.costos_camino[1] for hormiga in self.pareto_set]) / hormigas_optimas
            tau0_prima = 1 / (promedio_costo_1 * promedio_costo_2)
            global tau0
            if tau0 < tau0_prima:
                global tau
                tau0 = tau0_prima
                tau = [[tau0_prima if i != j else 0 for i in range(n)] for j in range(n)]
            else:
                for hormiga in self.pareto_set:
                    for idx in range(1, len(hormiga.camino)):
                        i = hormiga.camino[idx-1]
                        j = hormiga.camino[idx]
                        tau[i][j] = tau[j][i] = (1 - rho) * tau[i][j] + rho / (costo1[i][j] * costo2[i][j])

        def draw(self, pareto_set, subplot=111):
            """
```

```
            Dibuja el frente pareto.

            @param subplot: posiciÃ³n del grÃ¡fico.
            """
            fig = plt.figure()
            pf_ax = fig.add_subplot(subplot)
            pf_ax.set_title(u"Frente Pareto")
            for p in pareto_set:
                pf_ax.scatter(p.costos_camino[0], p.costos_camino[1], marker='o', facecolor='blue')
            plt.show()

        def distancia(self, a, b):
            dist = 0
            for i in range(len(a.costos_camino)):
                dist += (a.costos_camino[i]-b.costos_camino[i]) ** 2
            return math.sqrt(dist)

        def m1(self, y_true, frente_patero):
            suma = 0
            for p in frente_patero:
                dist = []
                for y in y_true:
                    dist.append(self.distancia(p, y))
                suma = suma + min(dist)
            return suma / len(frente_patero)

        def m2(self, sigma, frente_pareto):
            suma = 0
            for h1 in frente_pareto:
                for h2 in frente_pareto:
                    if h1 != h2 and self.distancia(h1,h2) > sigma:
                        suma = suma + 1
            if len(frente_pareto) - 1 > 0:
                resultado = suma / (len(frente_pareto) - 1)
            else:
                resultado = suma
            return resultado

        def m3(self, frente_pareto):
            dist_x = []
            dist_y = []
            n = len(frente_pareto)
            pareto = frente_pareto
            for i in range(n - 1):
                for j in range(i + 1, n):
                    dist_x.append((pareto[i].costos_camino[0] - pareto[j].costos_camino[0]) ** 2)
                    dist_y.append((pareto[i].costos_camino[1] - pareto[j].costos_camino[1]) ** 2)
            return math.sqrt(max(dist_x) + max(dist_y))

        def error(self, frente_pareto, y_true):
            interseccion = []
            for i in frente_pareto:
                agregado = False
                for j in y_true:
                    if i.costos_camino[0] == j.costos_camino[0] and i.costos_camino[1] == j.costos_camino[1] and
i.camino == j.camino and not agregado:
                        interseccion.append(i)
                        agregado = True
            return 1 - len(interseccion) / len(frente_pareto)


    class Hormiga:
        def __init__(self, num):
            self.num = num
            self.camino = []
            self.costos_camino = [[], []]

        def construir_camino_sin_feromonas(self):
            self.camino = []
            for i in range(len(self.costos_camino)):
                self.costos_camino[i] = 0

            lambd = self.num / m
            ini = int(os.urandom(1)[0] / 255 * 100)
            self.camino.append(0)
            for _ in range(1, n):
                pos_sig = [i for i in range(n)]
                for visited in self.camino:
                    pos_sig.remove(visited)
                q = os.urandom(1)[0] / 255
                i = self.camino[-1]
                if q <= q0:
                    max_prob = (-1, -1)
                    for j in pos_sig:
                        prob_actual = (eta1[i][j] ** (lambd * beta)) * (eta2[i][j] ** ((1 - lambd) * beta))
                        if max_prob[1] < prob_actual:
                            max_prob = (j, prob_actual)
                    sig_ciudad = max_prob[0]
                else:
                    suma_vecinos = 0
                    tau_temporal = {}
                    for j in pos_sig:
                        t = (eta1[i][j] ** (lambd * beta)) * (eta2[i][j] ** ((1 - lambd) * beta))
                        tau_temporal.update({j: t})
                        suma_vecinos += t
                    probabilidades = [tau_temporal[camino]/suma_vecinos for camino in tau_temporal]
```

```
                    sig_ciudad = np.random.choice(list(tau_temporal.keys()), p=probabilidades)

                self.camino.append(sig_ciudad)
                self.costos_camino[0] += costo1[i][j]
                self.costos_camino[1] += costo2[i][j]
                tau[self.camino[-2]][self.camino[-1]]  =  tau[self.camino[-1]][self.camino[-2]]  =  (1  -  rho)  *
tau[self.camino[-2]][self.camino[-1]] + rho * tau0

            tau[0][self.camino[-1]] = tau[self.camino[-1]][0] = (1 - rho) * tau[self.camino[-2]][self.camino[-1]] +
rho * tau0
            self.camino.append(0)

        def construir_nuevo_camino(self):
            self.camino = []
            for i in range(len(self.costos_camino)):
                self.costos_camino[i] = 0
            lambd = self.num / m
            ini = int(os.urandom(1)[0] / 255 * 100)
            self.camino.append(0)
            for _ in range(1, n):
                pos_sig = [i for i in range(n)]
                for visited in self.camino:
                    pos_sig.remove(visited)
                q = os.urandom(1)[0] / 255
                i = self.camino[-1]
                if q <= q0:
                    max_prob = (-1, -1)
                    for j in pos_sig:
                        prob_actual = tau[i][j] * (eta1[i][j] ** (lambd * beta)) * (eta2[i][j] ** ((1 - lambd) *
beta))
                        if max_prob[1] < prob_actual:
                            max_prob = (j, prob_actual)
                    sig_ciudad = max_prob[0]
                else:
                    suma_vecinos = 0
                    tau_temporal = {}
                    for j in pos_sig:
                        t = tau[i][j] * (eta1[i][j] ** (lambd * beta)) * (eta2[i][j] ** ((1 - lambd) * beta))
                        tau_temporal.update({j: t})
                        suma_vecinos += t
                    probabilidades = [tau_temporal[camino]/suma_vecinos for camino in tau_temporal]
                    sig_ciudad = np.random.choice(list(tau_temporal.keys()), p=probabilidades)
                self.camino.append(sig_ciudad)
                self.costos_camino[0] += costo1[i][j]
                self.costos_camino[1] += costo2[i][j]
                tau[self.camino[-2]][self.camino[-1]]  =  tau[self.camino[-1]][self.camino[-2]]  =  (1  -  rho)  *
tau[self.camino[-2]][self.camino[-1]] + rho * tau0

            tau[0][self.camino[-1]] = tau[self.camino[-1]][0] = (1 - rho) * tau[self.camino[-2]][self.camino[-1]] +
rho * tau0
            self.camino.append(0)

    print("KROAB100.TSP")
    import copy


    def test_metricas_moacs(file, y_true_nsga):
        valores = file

        global costo1, costo2, n, eta1, eta2, tau, ciudades

        costo1 = [[float(i) for i in j] for j in valores[2][:100]]  # Costo 1 entre las ciudades
        costo2 = [[float(i) for i in j] for j in valores[3][:100]]  # Costo 2 entre las ciudades

        n = len(costo1)
        eta1 = [[0 for i in range(n)] for j in range(n)]  # Visibilidad de los arcos para el objetivo 1
        eta2 = [[0 for i in range(n)] for j in range(n)]  # Visibilidad de los arcos para el objetivo 2
        tau = [[tau0 for i in range(n)] for j in range(n)]  # Feromonas de los arcos

        ciudades = [i for i in range(n)]
        for ciudad in ciudades:
            for vecino in ciudades[ciudad + 1:]:
                eta1[ciudad][vecino] = eta1[vecino][ciudad] = 1 / costo1[ciudad][vecino]
                eta2[ciudad][vecino] = eta2[vecino][ciudad] = 1 / costo2[ciudad][vecino]
                tau[ciudad][vecino] = tau[vecino][ciudad] = tau0

        frentes_pareto = []
        y_true_calculada = y_true_nsga
        for _ in range(5):
            col = Colonia()
            bestSoFar = copy.deepcopy(col.pareto_set)
            iters = 1
            stop = 250
            bestIter = 0

            while iters <= stop:
                # print(iters)
                col.nuevos_caminos()
                col.actualizar_frente_pareto()
                col.actualizacion_global_feromonas()
                conjunto_pareto = copy.deepcopy(col.pareto_set)
                iters += 1
            frentes_pareto.append(copy.deepcopy(conjunto_pareto))
            y_true_calculada = set(y_true_calculada).union(set(conjunto_pareto))
            y_true_calculada = col.encontrar_no_dominados(y_true_calculada)
            col.draw(col.pareto_set)
```

```
    col.draw(y_true_calculada)

    return y_true_calculada, frentes_pareto, col


def evaluar_moacs(y_true_calculada, frentes_pareto, col):
    m1 = []
    m2 = []
    m3 = []
    error = []
    for frente in frentes_pareto:
        m1.append(col.m1(y_true_calculada, frente))
        m2.append(col.m2(5000, frente))
        m3.append(col.m3(frente))
        error.append(col.error(frente, y_true_calculada))

    m1_prom = sum(m1)/len(m1)
    m2_prom = sum(m2)/len(m2)
    m3_prom = sum(m3)/len(m3)
    error_prom = sum(error)/len(error)

    print('M1 promedio: ', m1_prom)
    print('M2 promedio: ', m2_prom)
    print('M3 promedio: ', m3_prom)
    print('Error promedio: ', error_prom)

def y_true_nsga_a_moacs(y_true):
    y_true_moacs = []
    i = 1
    for ind in y_true:
        y_true_moacs.append(individuo_a_hormiga(ind, i))
        i += 1
    return y_true_moacs


def individuo_a_hormiga(individuo, i):
    h = Hormiga(i)
    h.camino = individuo.cromosoma
    h.costos_camino[0] = individuo.f1
    h.costos_camino[1] = individuo.f2
    return h


def y_true_moacs_a_nsga(y_true):
    y_true_nsga = []
    for h in y_true:
        y_true_nsga.append(hormiga_a_individuo(h))
    return y_true_nsga


def hormiga_a_individuo(h):
    i = Individuo(h.camino, costo1, costo2)
    i.cromosoma = h.camino
    i.f1 = h.costos_camino[0]
    i.f2 = h.costos_camino[1]
    return i
```

Resultados Implementados

Hardware utilizado

- Procesador I5 8va
- SSD 480GB Kingston
- Memoria 8 RAM

Resultados de las comparaciones

Primera Instancia

Evaluación

NSGA

Distancia al frente Pareto: 30121.190062433332

Distribución del frente Pareto: 4.166666666666667

Extensión del frente Pareto: 268.8950561049837

Error: 1.0
MOACS
M1 promedio: 10054.762689575575
M2 promedio: 8.107246376811593
M3 promedio: 64295.11100621791
Error promedio: 0.7085714285714285

Segunda Instancia
NSGA
MOACS
Evaluación
NSGA
Distancia al frente Pareto: 11048.773776533588
Distribución del frente Pareto: 3.12
Extensión del frente Pareto: 246.90738690236253
Error: 1.0
MOACS
M1 promedio: 10071.847772507552
M2 promedio: 10.798601398601397
M3 promedio: 100413.25625838686
Error promedio: 0.74491341991342

Conclusión

En el problema del Agente Viajero es un problema donde la solución ha sido estudiada desde los inicios de la Inteligencia Artificial considerando que su aplicación puede ser en cualquier área de estudio cuyos problemas reflejen una situación donde se tienen diferentes puntos a visitar con un costo considerado en el enlace entre dichos puntos. Cada autor ha propuesto soluciones para ciertas instancias de TSP, cada uno con una perspectiva diferente empleando técnicas que no son repetibles pero que, en determinado momento, se pueden emplear para dar lugar a nuevas soluciones

Trabajos Futuros

Realizar más comparaciones entre los algoritmos de MOACO y MOEA para cada tipo de problema ya sea TSP, QAP o hasta incluso VRPTW, también Llegar a utilizar más métricas para realizar la comparación entre los algoritmos, entre ellos se podría mencionar HiperVolumen, Epsilon, GeneralizedSpread, GeneralizationalDistance, InvertedGenerationalDistance, Spread (Métricas que se encontraron en el Framework de JMetal), entre otras.

Referencias

1. https://www.uaeh.edu.mx/scige/boletin/tlahuelilpan/n3/e5.html
2. https://baobabsoluciones.es/blog/2019/12/04/algoritmo-aco/
3. https://www.uaeh.edu.mx/scige/boletin/tlahuelilpan/n3/e5.html#:~:text=TSP%20es%20un%20problema%20considerado,un%20tiempo%20de%20c%C3%B3mputo%20razonable.
4. https://slideplayer.es/slide/24026/