Facultad de Ciencias, UNAM
Resolución al problema de obtener puntos Steiner usando la heuristica de PSO.

Amézcua Lopez, Marisol

10 de diciembre de 2018

1. Introducción

El problema del árbol de Steiner (STP Steiner Tree Problem por sus siglas en inglés) es un problema de optimización combinatoria que también entra en el área de geometría. Este problema trata de dados n puntos encontrar la mejor red para ellos, en otras palabras busca un árbol tal que la suma de sus aristas sea mínima. Puntos extras llamados puntos Steiner también pueden ser agregados.

Este problema puede ser visto en el plano euclideano n dimensional o bien en gráficas con pesos en las aristas. Este problema tiene varias variantes y aplicaciones como en problemas de enrutamiento, diseño de redes, biología computacional, etc[3].

La propuesta que se verá en este artículo para resolver el problema es usar el algoritmo de Particle Swarm Optimization con ciertas modificaciones.

2. Definición del problema

En la geometría existe un árbol llamado *árbol generador mínimo euclideano* (EMST por sus siglas en inglés), el cual es dado un conjunto de puntos S en el plano cumple que la suma de las aristas que conectan todos estos puntos es mínima.

El EMST se puede mejorar si se le agregan nuevos puntos de conexión llamados *puntos Steiner* minimizando aún mas la suma de las aristas. Sin embargo en 1976 Garey, Graham y Johnson demostraron que este problema era NP-duro y no era viable resolver el problema con mas de 20-25 puntos[2]. Es por esto que en este artículo se buscará una manera para encontrar estos puntos sin usar un algoritmo exacto.

3. Descripción de la propuesta e implementación

El modelo de optimización por enjambre de partículas (PSO por sus siglas en inglés Particle Swarm Optimization) es una estrategia de optimización basada en la en la estrategia de Kennedy y Eberhart

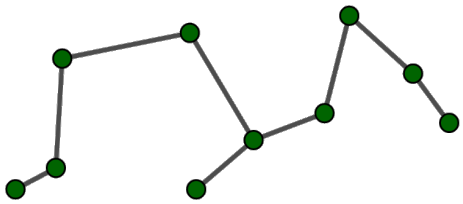

Figura 1: Ejemplo de un EMST

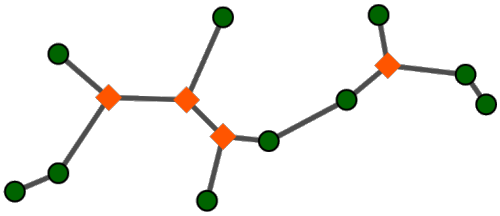

Figura 2: Un árbol Steiner dados los mismos puntos.

(1995). Para encontrar la solución óptima de un problema, dado un conjunto de partículas cada partícula ajusta su vuelo (o bien velocidad y dirección) de acuerdo a la experiencia propia y a sus compañeros. El algoritmo se suele separar en la parte cognitiva y la parte social, la parte social esta dada por lo que se llamo g_best (se obtiene la experiencia de vuelo de todos los compañeros o bien vecinos) y l_best (lo cual es la experiencia de vuelo de los vecinos más cercanos o bien con cierto criterio)[4].

3.1. El algoritmo PSO básico.

El algoritmo consiste en un conjunto de partículas que se mueven en un plano n-dimensional, este será el espacio de búsqueda de posibles soluciones. Para la búsqueda se define una función fitness que evalúe la aptitud de las partículas, para así poder comparar las soluciones que tenemos. Cada partícula i en el tiempo t cuenta con lo siguiente:

- $X_{i,t}$, $V_{i,t}$ son los vectores de posición y velocidad.
- $P_{i,t}$ es un apuntador al historial de la partícula que guarda la posición del "mejor fitness.$^{\text{en}}$contrado.
- $G_{i,t}$ es la mejor posición global (esta no siempre se tiene).
- $C_{i,t}$ es la evaluacion del fitness actual dada la posicion actual
- $C_{b_{i,t}}$ es la evaluacion del fitness del mejor local encontrado.

En cada iteración t la velocidad se actualiza y la partícula se mueve a una nueva posición. Esta nueva posición se calcula con lo siguiente:

$X_{i,t+1} = X_{i,t} + V_{i,t+1}$

Y la velocidad se actualiza con lo siguiente:

$V_{i,t+1} = c_1 \cdot V_{i,t} + c_2 \cdot r_2 \cdot (P_{i,t} - X_{i,t}) + c_3 \cdot r_3 \cdot (G_{i,t} - X_{i,t})$

c_1 es una constante introducida por Shi y Eberhart (1998) es el peso de inercia, en otras palabras es una constante que evita que la velocidad se incremente indefinidamente, esto controla la magnitud de la velocidad anterior. Los parámetros c_2 y c_3 son números reales escogidos de forma uniforme en un intervalo aleatorio $[0, 1]$. Estos valores determinan que tanto se desvía la partícula con respecto al mejor global o a la información local. Los valores r_2 y r_3 son valores que se ajustan de acuerdo al problema, nótese que estos afectan a la importancia que le damos a seguir al líder o a regresar al óptimo de la partícula. Las 3 componentes de la velocidad se pueden describir de la siguiente manera:

- $c_1 \cdot V_{i,t}$ sirve como el término de momentum para evitar el incremento de la velocidad.
- $c_2 \cdot r_2 \cdot (P_{i,t} - X_{i,t})$ representa el componente cognitivo, representa la tendencia natural de los individuos a regresar a un ambiente donde experimentaron una mejor experiencia.
- $c_3 \cdot r_3 \cdot (G_{i,t} - X_{i,t})$ representa el componente social, representa la tendencia de los individuos a seguir al líder o bien el éxito de sus compañeros.

La vecindad puede ser cambiada de acuerdo a diferentes criterios, dado estos se definen los vecinos de la partícula para poder tener una componente local.

El pseudocódigo del algoritmo sería el siguiente:

Data:
Result:
Inicializar parámetros del PSO, coeficientes y el tamaño de la población;
Inicializar posiciones y velocidades iniciales de las partículas;
Inicializar el mejor global y el mejor para cada partícula y vecindad;
while *no se alcance iteracion máxima o claúsula de escape* **do**
    actualizar velocidad;
    actualizar posiciones;
    determinar el mejor para cada partícula;
    determinar el individuo élite ;
end

Algorithm 1: Pseudocódigo de un PSO básico

Para este proyecto en particular se utilizará el PSO ejecutado varias veces para intentar encontrar la mayor cantidad de puntos para minimizar el costo del arbol lo mas posible.

Una vez que tenemos el concepto del algoritmo claro procederemos a relacionarlo con resolver el problema del árbol de Steiner. La implementación recibirá como entrada un conjunto de puntos $\mathcal{S}$ y el primer paso será calcular un árbol de peso mínimo, los pesos de las aristas corresponderán a la distancia entre los puntos y los vértices serán los puntos. Para esto se pueden utilizar algoritmos conocidos como Prim o Kruskal.

Una vez con esta gráfica comenzamos a usar PSO, generamos un enjambre, esperamos a que termine de optimizar y después lanzamos el siguiente enjambre. Para generar los k podemos dejar un número fijo de repeticiones.º bien hasta que dejemos de optimizar. Entonces para cada iteración inicializamos las m partículas para cada enjambre con sus posiciones iniciales (para esto se usará el punto donde queremos generar el sub-enjambre) y velocidades iniciales (Para esto usaremos números aleatorios), actualizamos el mejor del enjambre y el mejor para cada partícula.

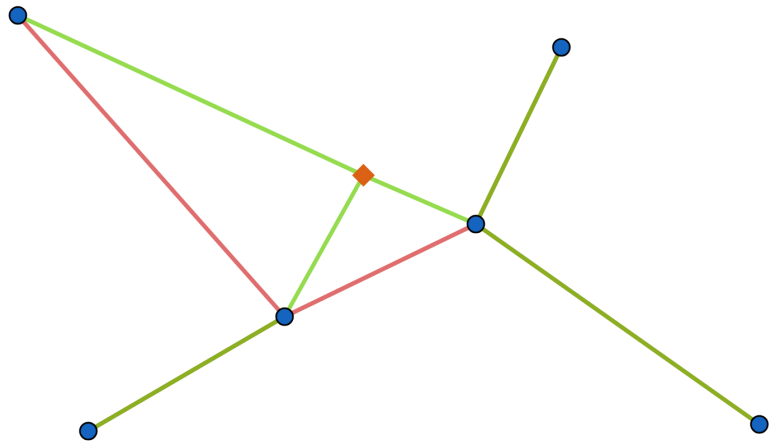

Figura 3: El punto naranja representa la partícula, el árbol verde es el árbol mínimo que incluye a la partícula y el rosa (el cual esta superpuesto con el verde) representa al árbol de peso mínimo original.

Una vez con esto entramos al ciclo que indica el PSO y realizamos las operaciones para cada enjambre. Para esto se necesitará la función fitness y un criterio para matar partículas cuando dejan de optimizar, esto con el fin de ahorrar recursos computacionales. La función **fitness** será la suma de las distancias, es decir, las aristas del árbol obtenido cuando agregamos la partícula a evaluar al árbol generado al inicio. Para esto conectaremos a la posición de la partícula (la cual es un punto) con todos los puntos del conjunto $\mathcal{S}$ y usando Prim obtendremos el árbol asociado, el cual tiene la característica de que conecta a la partícula con sus puntos más cercanos y da paso a una optimización de la suma de las aristas.

Resumiendo el pseudocódigo de nuestro algoritmo:

Data: Un conjunto $\mathcal{S}$ de puntos
Result: Un conjunto de puntos $\mathcal{V}$
Inicializar $\mathcal{V}$ **while** *no se alcance k máxima o ya no haya optimización* **do**
    Inicializar parámetros del PSO, coeficientes y el tamaño de la población;
    Inicializar posiciones y velocidades iniciales de las partículas;
    Inicializar el mejor global y el mejor para cada partícula y vecindad;
    while *no se alcance iteracion máxima o claúsula de escape* **do**
        actualizar velocidad;
        actualizar posiciones;
        determinar el mejor para cada partícula;
        determinar el individuo élite ;
    end
    $\mathcal{V} \bigcup \{g_{best}\}$
end

Algorithm 2: Pseudocódigo del algoritmo

3.2. Implementación

Para el proyecto se utilizó una implementación en julia, donde recibimos un conjunto de puntos. Posteriormente creamos una gráfica completa con esos puntos y su árbol de peso minimo donde los pesos de las aristas estan dados por la formula de la distancia entre dos puntos euclideanos.

```
"Funcion que contruye el arbol dado el conjunto de puntos obtenidos"
function construye_arbol(S)
        S = map(x -> float(x),S) #Pasamos todos los elementos a flotantes
        long = size_set(S)
        puntos = Dict{Int64, Array{Float64,1}}() #Creamos un diccionario para
        #guardar los puntos
        i = 1
        for v in S
                puntos[i] = v
                i +=1
        end
        G = SimpleWeightedGraph(long) #Hacemos una grafica con
         la cantidad de vertices igual a la cantidad de puntos
        for v in puntos
                for u in puntos
                        if u != v
                                add_edge!(G,u[1],v[1],
                                distancia_2_puntos(u[2],v[2])) #Conectamos
                                 #a todos los vertices con los demas
                        end
                end
        end
        #En este punto tenemos la grafica completa
        aristas_arbol = kruskal_mst(G)
        Arbol = SimpleWeightedGraph(long)
        for v in aristas_arbol
                add_edge!(Arbol,v.src, v.dst, v.weight)
        end

        return Arbol, puntos #Regresamos el arbol y el diccionario
end
```

Una vez con esto calculamos el peso del árbol y lo guardamos en una variable para no tener que recalcular:

```
"Funcion que obtiene el peso total de un arbol"
function obten_peso_total(arbol)
        suma = 0
        for v in edges(arbol[1])
                suma += v.weight
        end
```

```
        return suma
end
```

Así pues iniciamos los enjambres, al algoritmo le pasamos la cantidad de enjambres y el tamaño de población. Este por otra parte ejecuta el algoritmo PSO básico:

```
"Funcion que se encarga de crear los swarms y buscar
puntos Steiner en la grafica"
function obten_puntos_steiner(st::Steiner,
iteracion_maxima::Int64,swarms::Int64,
tam_pob::Int64,nombre_archivo)
        k = 0
        peso_0 = st.peso
        peso_inicial = st.peso
        semilla = abs(rand(Int))#Para reproduccion de resultados
        Random.seed!(semilla)
        while k < swarms
                x1 = Float64(rand(particula.limite_inferior[1]:
                 particula.limite_superior[1]))
                x2 = Float64(rand(particula.limite_inferior[2]:
                 particula.limite_superior[2]))
                ps = pso.PSO(eval_arbol,tam_pob,[x1,x2])
                punto_steiner = pso.corre_pso(iteracion_maxima,ps)
                punto = punto_steiner.mejor_posicion
                fit_actual = punto_steiner.fitness
                if peso_0 > punto_steiner.fitness
                        porcentaje = 1 - punto_steiner.fitness/peso_0
                        global arbol_actual = obten_arbol_con_
                        punto(arbol_actual,punto)
                        peso_0 = punto_steiner.mejor_fitness
                end
                k+=1
        end
end
```

En la implementación se cuenta con que cada partícula conserva la posición de su mejor fitness además de el valor del fitness, esto es para poder hacer mejor las comparaciones y no volver a calcular el árbol.

También se cuenta con un criterio para matar y crear partículas si es que esta lleva estancada un rato, así como también se cuenta con un criterio para detener la ejecución de un enjambre si es que este deja de mejorar.

```
"Funcion que se encarga de verificar si debemos matar a una particula dado un
 numero maximo para empeorar,
si la mata debe de generar otra con alguna posicion"
function debo_matar(particula::Particula, empeora_max::Int64)
```

```
        if particula.empeora > empeora_max
                #Creamos una nueva particula
                p = Particula(particula.dimension,
                particula.posicion, particula.fun_fitness)
                particula.mejor_fitness = p.mejor_fitness
                particula.mejor_posicion = p.mejor_posicion
                particula.velocidad = p.velocidad
                particula.empeora = 0
                particula.fitness = p.fitness
                particula.posicion = p.posicion
        end
        return particula
end
```

Otro punto importante es la función fitness, la cual agrega el punto a la gráfica con el resto de los nodos, posteriormente obtiene el árbol y después obtiene el peso total del árbol. Este peso total será el valor del fitness.

```
# Funcion que devuelve el fitness de un punto
function eval_arbol(p)
        G= obten_arbol_con_punto(arbol_actual,copy(p))
        peso = obten_peso_total(G)
        return peso
end
```

3.3. Ejemplo de ejecución

Tomemos el conjunto $\mathcal{S}$ = [[2.0, 5.0], [2.0, 8.0], [-2.0, 5.0], [-2.0, 8.0]], el peso del árbol original es de *10.0*, al correr el algoritmo este nos devuelve los puntos Steiner junto con los originales y el peso del árbol obtenido, siendo así el conjunto de puntos Steiner:

[-1.30421, 7.24882],[1.18885, 6.93253],[-1.04213, 6.83676],[1.0764, 6.75418],[-0.992491, 6.75109],[1.052, 6.71366],[-0.981047, 6.73087]

Y el peso final 9.233380800612336 con un porcentaje de mejora de 7.6 % aproximadamente.

Si lo graficamos obtenemos un árbol como el siguiente:

Despues de obtener los puntos Steiner al graficar el arbol de peso minimo tenemos lo siguente:

Un ejemplo mas complicado con el siguiente conjunto $\mathcal{S}$ = [[-37.0, 91.0], [-80.0, 68.0], [65.0, 98.0], [60.0, -87.0], [-67.0, -62.0], [-84.0, 22.0], [-71.0, 6.0], [82.0, -96.0], [-10.0, 58.0], [76.0, -25.0], [48.0, -7.0], [-31.0, 54.0], [-59.0, -39.0], [40.0, -64.0], [-75.0, -8.0], [-19.0, 6.0], [67.0, 55.0], [-31.0, -95.0], [-53.0, -94.0], [-75.0, -83.0], [-70.0, 87.0], [-16.0, -13.0], [58.0, -14.0], [52.0, -61.0], [-54.0, 96.0], [-9.0, 12.0], [-37.0, -20.0], [77.0, 13.0], [79.0, -4.0], [-5.0, 67.0], [-36.0, 18.0], [88.0, 86.0], [-16.0, -59.0], [35.0, -38.0], [16.0, -40.0], [-99.0, 67.0], [22.0, 65.0], [-93.0, 50.0], [48.0, -46.0], [-39.0, -69.0]] la grafica original se ve a continuacion en la figura 6.

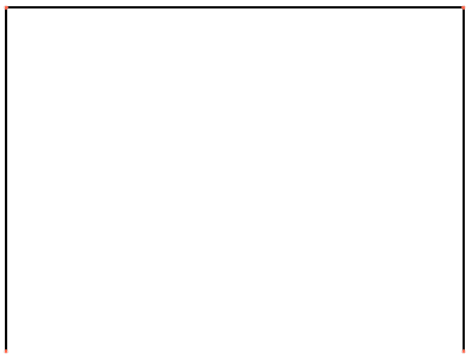

Figura 4: Grafica original con los 4 puntos iniciales.

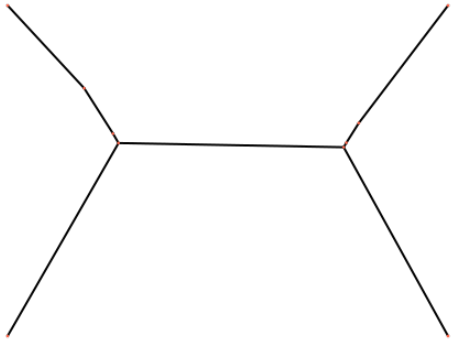

Figura 5: Arbol final obtenido al incluir los puntos steiner encontrados.

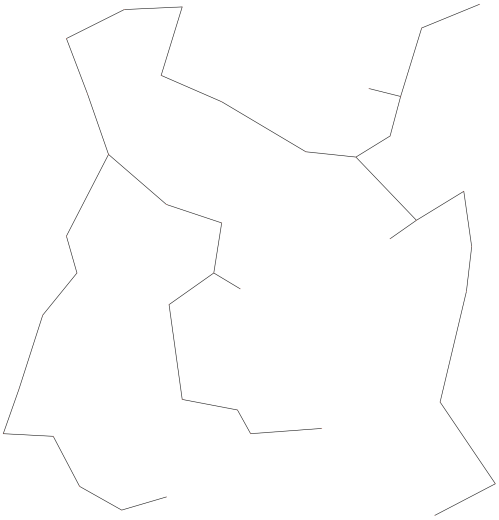

Figura 6: Grafica original antes de correr el algoritmo.

Su peso original es de *899.688517537882*. Despues de varias iteraciones se obtuvieron los siguientes puntos Steiner:

[-33.3787, -71.5279],[-55.9436, -25.7838],[71.3177, -15.0948],[16.0051, -39.9995],[-25.5842, 60.5872],[77.1127, 83.9357],[49.4163, -64.751],[43.6034, -38.1243],[53.6567, -14.5274],[-69.7045, -80.8769],[-37.3532, -89.133],[-92.0509, 62.2354],[-19.592, -10.7534],[43.0271, -37.3007],[-34.9562, -72.1639],[74.8999, 86.505],[49.7796, -52.6736],[-4.64425, 66.3075],[-18.9513, 6.92918],[-10.3181, 58.7451],[-31.4452, 15.0423],[-25.2502, 61.0056],[-4.70283, 66.3143],[-74.4707, -7.84893]

Con un peso final de *873.3105296213629* y un porcentaje de mejora total de *2.93 %*. La grafica final es la siguiente.

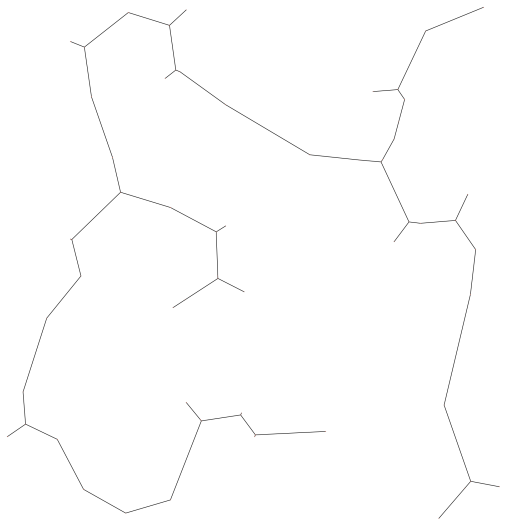

Figura 7: Grafica obtenida despues de correr el algoritmo.

Referencias

[1] Russel, S, & Norvig, P.. (2010). *Artificial Intelligence A Modern Approach*. New Jersey: Pearson.

[2] Preparata, Franco P. & Shamos, M. . (1985). Computational Geometry. Springer-Verlag New York: Springer.

[3] Decroos, T., De Causmaecker, P. & Demoen, B.. (2015). Solving Euclidean Steiner Tree Problems with Multi Swarm Optimization. En Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation(pp.1379-180). Madrid, España: ACM.

[4] Sylverin Kemmoé Tchomté & Gourgand, M.. (Mayo 2009). Particle swarm optimization: A study of particle displacement for solving continuous and combinatorial optimization problems. International Journal of Production Economics, 121, pp.57-67.

[5] Claude Sammut & Geoffrey I. Webb. (2011). PAC Learning. En Encyclopedia of Machine Learning(pp.745-817). US: Springer Science.