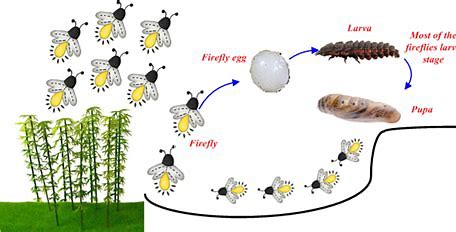
# Giới thiệu chung



Thuật toán Firefly Algorithm Optimization (FAO) là một phương pháp tối ưu hóa tiến hóa dựa trên mô phỏng cách mà côn trùng đom đóm (firefly) tương tác và thu hút nhau trong không gian tìm kiếm. Nó được đề xuất bởi X.-S. Yang vào năm 2008 và đã được sử dụng rộng rãi trong các bài toán tối ưu hóa.

Ý tưởng chính của thuật toán FAO là mô phỏng cách côn trùng đom đóm sử dụng sự phát quang để thu hút và tìm đến những vị trí tốt hơn. Thuật toán bắt đầu với một quần thể các "đom đóm" (fireflies), mỗi cá thể được biểu diễn bằng một vector các tham số tối ưu hóa. Các cá thể sẽ di chuyển trong không gian tìm kiếm và tương tác với nhau dựa trên cường độ phát quang của chúng

# Ứng dụng

1. Tối ưu hóa các thông số trong mô hình kinh tế:

Các mô hình kinh tế phức tạp có thể có nhiều thông số cần được điều chỉnh để đạt được hiệu suất tối ưu. Firefly Algorithm có thể được sử dụng để tối ưu hóa các thông số này, như tối ưu hóa chi phí sản xuất, tối ưu hóa các mô hình dự báo, và tối ưu hóa phân bổ tài nguyên.

1. Mạng lưới điện

Trong hệ thống điện lực, việc phân phối tối ưu và lập lịch hoạt động của các đơn vị sản xuất và tiêu thụ điện có thể được thực hiện bằng cách sử dụng Firefly Algorithm để tối ưu hóa hiệu suất và giảm thiểu chi phí.

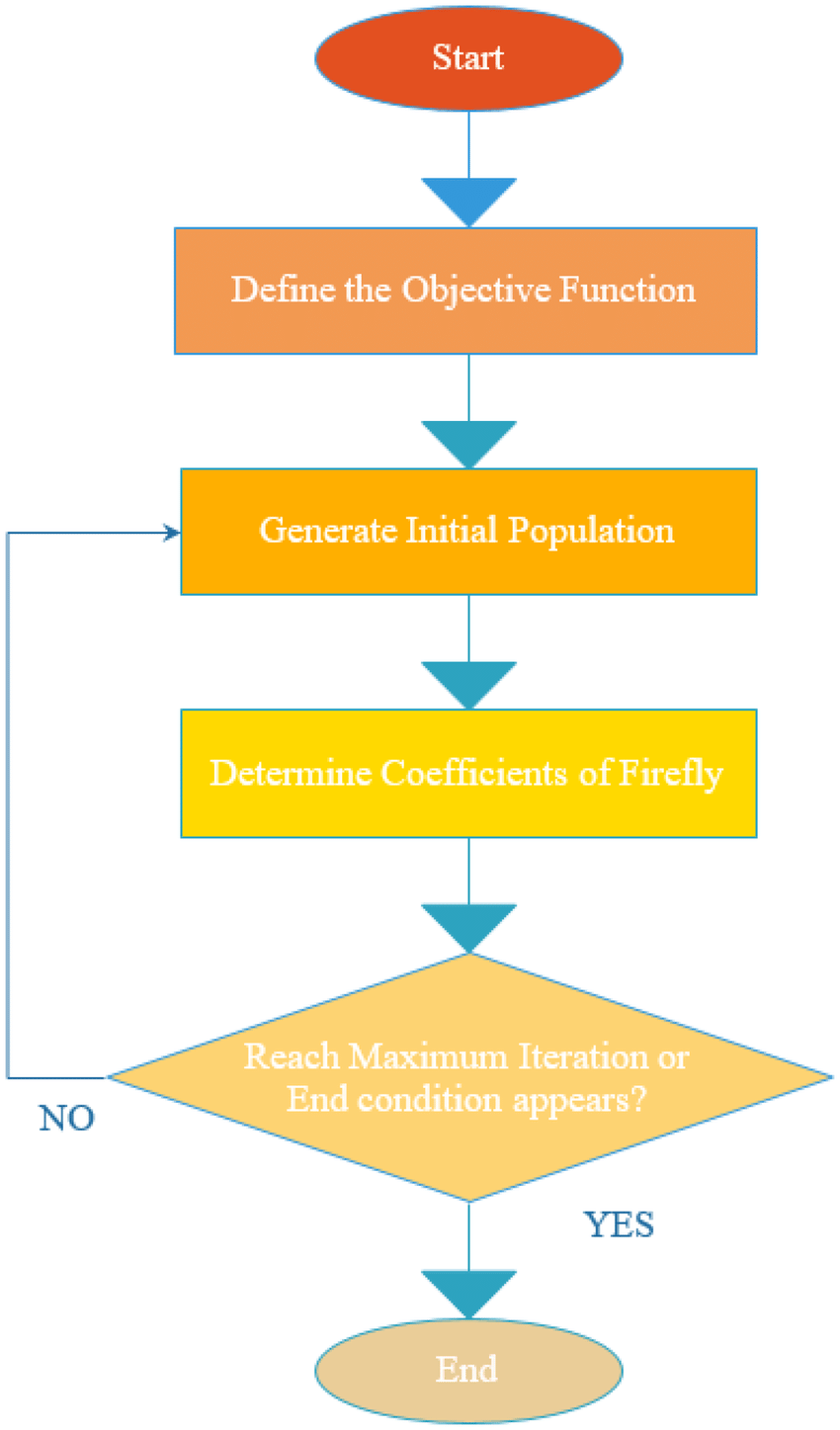
1. Tối ưu hóa các mạng và hệ thống phức tạp:

Firefly Algorithm có thể được sử dụng để tối ưu hóa cấu trúc và hiệu suất của các mạng máy tính, mạng xã hội, hệ thống giao thông, và hệ thống sản xuất.

1. Tối ưu hóa các bài toán tối ưu hóa tổ hợp:

Trong các bài toán tổ hợp với không gian tìm kiếm lớn và hàm mục tiêu phức tạp, Firefly Algorithm có thể cung cấp một giải pháp hiệu quả để tìm ra giải pháp gần tối ưu.

1. Mô tả chung của thuật toán



\*\*Bước 1: Khởi tạo\*\*

- Chọn số lượng đom đóm trong quần thể, ký hiệu là N.

- Đặt giá trị ban đầu cho các tham số của mỗi đom đóm, ví dụ như các giá trị ngẫu nhiên trong khoảng [min\_value, max\_value] của bài toán tối ưu.

\*\*Bước 2: Đánh giá\*\*

- Tính toán giá trị của hàm mục tiêu tại vị trí của từng đom đóm.

- Xác định đom đóm có giá trị tối ưu hóa tốt nhất trong quần thể, ký hiệu là x\_best.

\*\*Bước 3: Cập nhật vị trí\*\*

- Với mỗi đom đóm (cá thể), tính toán vector hướng di chuyển d, được tính bằng công thức:

d = x\_best - x\_i

Trong đó, x\_i là vị trí của đom đóm hiện tại.

- Tính toán khoảng cách r giữa hai đom đóm:

r = sqrt(sum(d^2))

- Tính toán sức mạnh thu hút A giữa hai đom đóm dựa trên khoảng cách r và một hệ số thu hút beta (β):

A = exp(-beta \* r^2)

- Nếu đom đóm kia có giá trị tối ưu hóa tốt hơn (có giá trị nhỏ hơn), thì sức mạnh thu hút A sẽ lớn hơn.

- Cập nhật vị trí mới của đom đóm (x\_new)

Trong đó, alpha (α) là hệ số điều chỉnh tốc độ di chuyển.

\*\*Bước 4: Kiểm tra điều kiện dừng\*\*

- Kiểm tra điều kiện dừng, ví dụ như số lượng vòng lặp đạt mức tối đa, giá trị tối ưu hóa đạt đến ngưỡng cụ thể hoặc thỏa mãn điều kiện dừng khác.

\*\*Bước 5: Lặp lại\*\*

- Nếu không đạt được điều kiện dừng, quay lại Bước 2 và tiếp tục tiến hành lặp lại quá trình cập nhật vị trí và đánh giá đến khi thoả mãn điều kiện dừng.

# Mô tả chi tiết vào bài toán thực tế (Travelling Salesman Problem (TSP)

Vấn đề người du lịch (TSP) trả lời câu hỏi sau: "Cho một danh sách các thành phố và khoảng cách giữa mỗi cặp thành phố, hãy tìm lộ trình ngắn nhất có thể để thăm tất cả các thành phố và quay trở lại thành phố xuất phát?"

Các bước thực hiện và code trong python

B1: Khởi tạo

-Mô hình hóa con đom đóm

class FireFly:

  def \_\_init\_\_(self,path,matrix\_weight,cost = None):

    self.path  = path

    self.matrix\_weight = matrix\_weight

    if cost:

      self.cost = cost

    else:

      self.cost = self.compute\_cost()

  def compute\_cost(self):

    cost = 0

    for i in range(len(self.path)-1):

      nodei = self.path[i]

      nodej = self.path[i+1]

      cost += self.matrix\_weight[nodei,nodej]

    return cost

-Khởi tạo N con đom đóm (mỗi con đom đóm là một lời giải) ngẫu nhiên

  def init\_quan\_the(self):

    paths = []

    quan\_the = []

    while(len(paths)<self.num\_firefly):

      new\_path = [i for i in range(1,self.num\_nodes)]

      random.shuffle(new\_path)

      new\_path = [0] + new\_path + [0]

      if new\_path not in paths:

        paths.append(new\_path)

        quan\_the.append(FireFly(new\_path,self.w\_matrix))

    return quan\_the

B2: Lấy lần lượt 2 cá thể I1 và I2 trong quần thể

Nếu lấy hết các cặp xuống B5

B3: So sánh độ sáng (quãng đường) nếu I1 có độ sáng nhỏ hơn I2(quãng đường dài hơn)

* Thực hiện bước 4
* Nếu không quay lại bước 2

B4:Tính khả năng hấp dẫn của I2 và I1 dựa theo công thức

Atraction = e^(-beta\*r)

beta là siêu tham số

r là khoảng cách hamming giữa 2 cá thể

  def atrative(self,ca\_the1:FireFly,ca\_the2:FireFly,beta = 0.3):

    r = hamming\_distance(ca\_the1,ca\_the2)

    return np.exp(-beta\*pow(r/self.num\_nodes,2))

Khoảng cách hamming là số vị trí khác nhau giữa 2 đường đi :

Ví dụ I1 = [0,1,2,3,4,0], I2 = [0,2,1,3,4,0] , thì r = 2

def hamming\_distance(

                       fire1:FireFly,

                       fire2:FireFly

                       ):

    value = (np.array(fire1.path)-np.array(fire2.path))!=0

    value = np.where(value,1,0)

    return np.sum(value)

Atraction thuộc khoảng từ 0 đến 1 và đây cũng chính là tỷ lệ để đột biến tạo ra các cá thể mới.Theo công thức khả năng đột biến tỷ lệ nghịch với khoảng cách.Nói cách khác càng gần thì càng dễ bị thu hút tạo đột biến .

Trong bài toán TSP 2 cá thể được coi là gần nhau khi thứ tự các điểm đến gần như là giống nhau.Đột biến như vậy sẽ tạo ra các cá thể mới và sẽ làm lên sự đa dạng sinh học.

Phép đột biến ở đây được dùng là phép đảo ngược . Ta sẽ chọn ngẫu nhiên điểm bắt đầu và chiều tỷ lệ thuận với r

 def inverse\_move(self,ca\_the:FireFly,r:int):

    if r<2 :

      r = 2

    length = random.randint(2,r)

    max\_begin = self.num\_nodes-length

    begin = random.randint(1,max\_begin)

    end = begin+length

    new\_ca\_the\_path = ca\_the.path.copy()

    new\_ca\_the\_path[begin:end] = reversed(new\_ca\_the\_path[begin:end])

    new\_ca\_the = FireFly(new\_ca\_the\_path,self.w\_matrix)

    return new\_ca\_the

  def mutation(self,ca\_the:FireFly,r:int):

    return [self.inverse\_move(ca\_the,r) for \_ in range(self.num\_mutation)]

Mỗi lần đột biến sẽ tạo ra M cá thể mới .

B6 : Chọn ra N cá thể đom đóm mới tốt nhất(đường đi ngắn nhất) trong N cá thể cũ+ K cá thể mới

B7: Kiểm tra điều kiện dừng

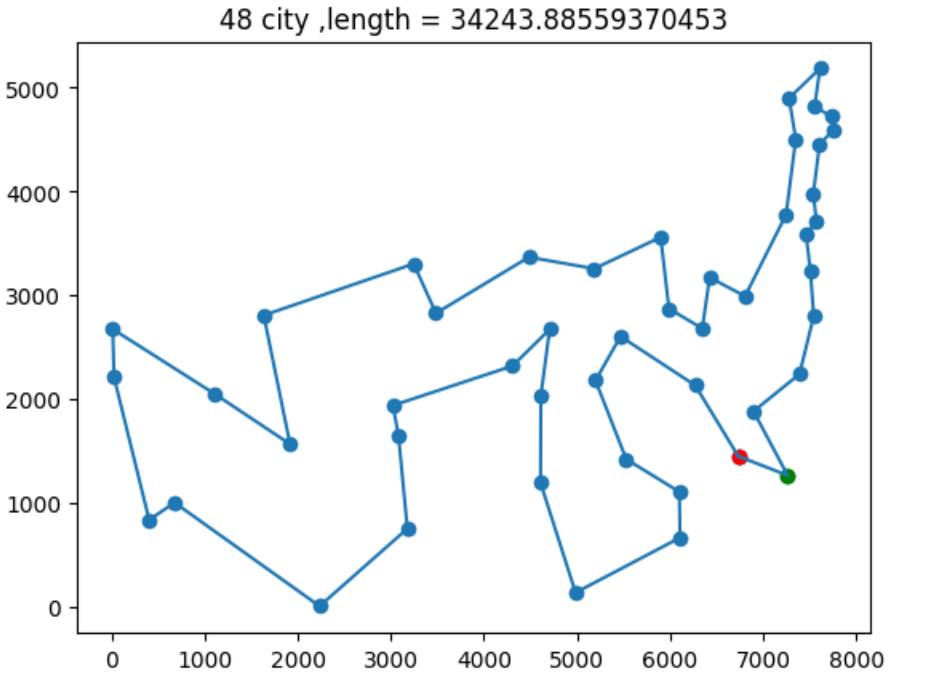
-Quãng đường ngắn nhất không thay đổi sau một số lần nhất định

- Vượt qua số lần lặp tối đa

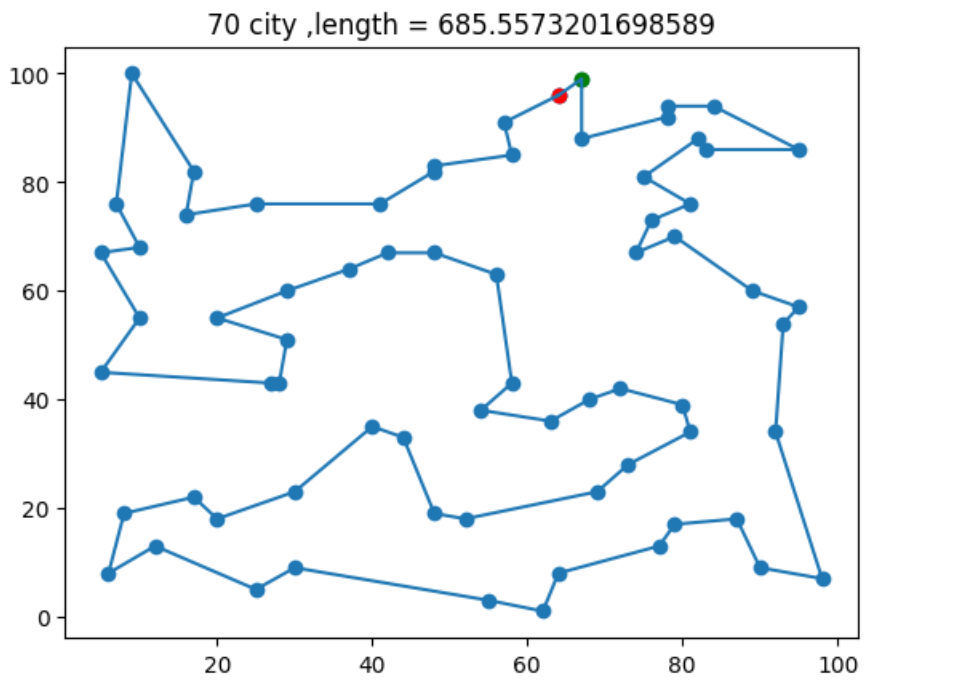
Nếu không quay lại bước 2

Full code: https://colab.research.google.com/drive/15GV3xdsX1StJIQ0z2GCPtF6ADGLzKDV5#scrollTo=A9FaeD9IbOjF

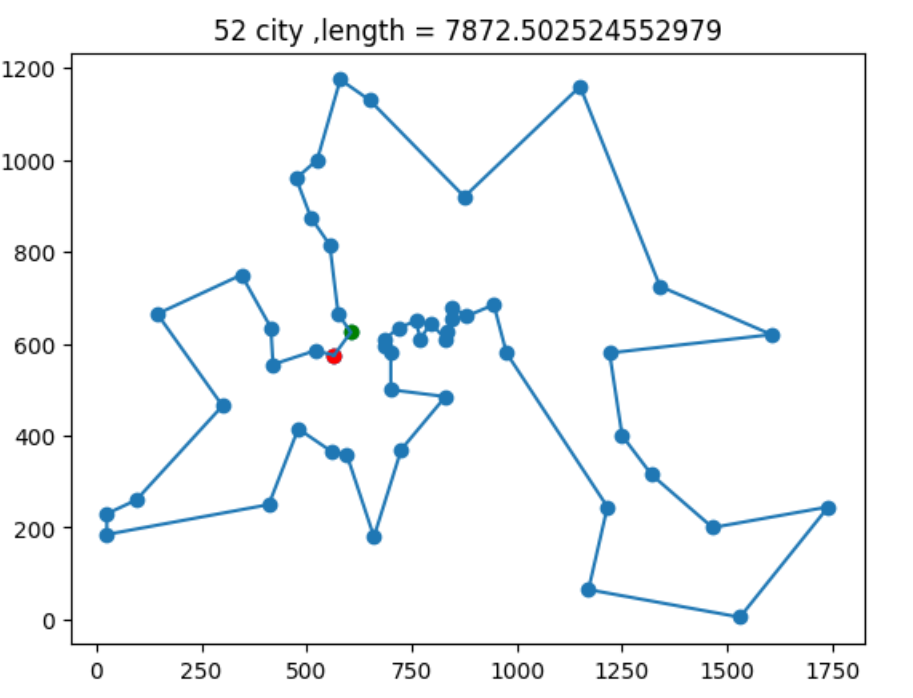
Trực quan hóa kết quả



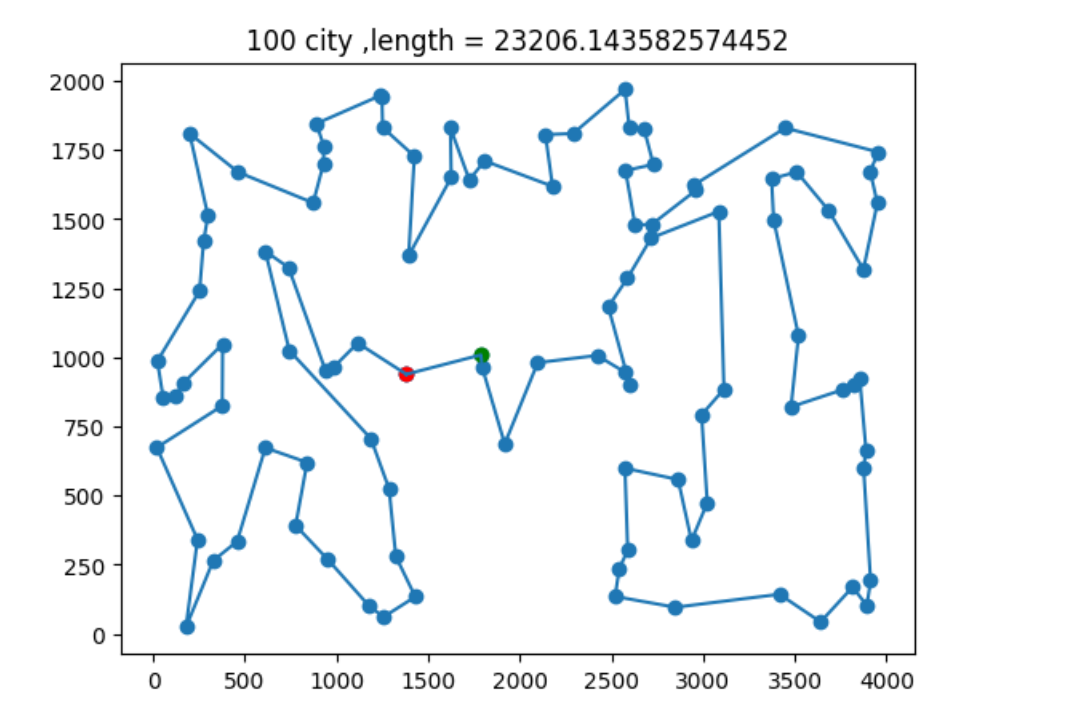
File dữ liệu att48.tsp



File dữ liệu st70.tsp



File berlin52.tsp



File kor100.tsp

Bảng số liệu so sánh trong đó

N là số lượng đom đóm ban đầu ,M là số lượng đột biến và sẽ lấy giá trị tốt nhất trong 10 lần chạy