**BÁO CÁO**

**Chủ đề: Giải quyết Bài toán Người du lịch (Traveling Salesman Problem) sử dụng các thuật toán Tìm kiếm cục bộ**

**Mục lục**

1. **Giới thiệu Bài toán Người du lịch (TSP)**
   * 1.1. Mục tiêu
   * 1.2. Không gian trạng thái
   * 1.3. Hàm mục tiêu
   * 1.4. Phép toán lân cận (Local Moves)
2. **Chuẩn bị và Dữ liệu thử nghiệm**
   * 2.1. Các thư viện và hàm hỗ trợ
   * 2.2. Thiết lập bài toán ngẫu nhiên
3. **Áp dụng các Thuật toán Tìm kiếm cục bộ**
   * 3.1. Giải pháp tham khảo sử dụng R
   * 3.2. Thuật toán Leo đồi Steepest-Ascend (Steepest-Ascend Hill Climbing)
   * 3.3. Thuật toán Leo đồi Steepest-Ascend với Khởi đầu ngẫu nhiên (Random Restarts)
   * 3.4. Thuật toán Leo đồi Stochastic (Stochastic Hill Climbing)
   * 3.5. Thuật toán Leo đồi First-Choice (First-Choice Hill Climbing)
   * 3.6. Thuật toán Luyện kim mô phỏng (Simulated Annealing)
   * 3.7. Thuật toán Di truyền (Genetic Algorithm)
4. **So sánh và Đánh giá Hiệu năng**
   * 4.1. Mã nguồn so sánh
   * 4.2. Kết quả thực nghiệm
   * 4.3. Phân tích kết quả
5. **Tổng kết và Đề xuất**
   * 5.1. Bảng tổng hợp
   * 5.2. Nhận xét và đề xuất

**1. Giới thiệu Bài toán Người du lịch (Traveling Salesman Problem - TSP)**

Bài toán Người du lịch là một trong những bài toán tối ưu hóa nổi tiếng nhất trong khoa học máy tính. Nó thuộc lớp bài toán NP-hard, nghĩa là không có thuật toán hiệu quả nào được biết đến để tìm ra lời giải tối ưu trong thời gian đa thức khi số lượng thành phố tăng lên.

**1.1. Mục tiêu**

Mục tiêu của bài toán là tìm ra một chu trình ngắn nhất đi qua tất cả các thành phố trong một danh sách cho trước, mỗi thành phố chỉ được đi qua đúng một lần và quay trở về thành phố xuất phát.

**1.2. Không gian trạng thái**

Mỗi trạng thái đại diện cho một chu trình (tour). Với N thành phố, một chu trình có thể được biểu diễn dưới dạng một hoán vị của các thành phố, cho biết thứ tự chúng được ghé thăm. Ví dụ, là thành phố đầu tiên, là thành phố thứ hai, và cứ thế tiếp diễn.

**1.3. Hàm mục tiêu**

Hàm mục tiêu cần được tối thiểu hóa là tổng độ dài của chu trình. Công thức tính như sau:

Trong đó di,j​ là khoảng cách giữa thành phố i và thành phố j.

**1.4. Phép toán lân cận (Local Moves)**

Để di chuyển từ trạng thái này sang trạng thái khác trong không gian tìm kiếm, phép toán lân cận cơ bản được sử dụng là **hoán đổi vị trí của hai thành phố** trong chu trình hiện tại.

**2. Chuẩn bị và Dữ liệu thử nghiệm**

**2.1. Các thư viện và hàm hỗ trợ**

Đầu tiên, chúng ta cần nhập các thư viện cần thiết và định nghĩa các hàm cơ bản để làm việc với bài toán TSP.

Python

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import math

import random

np.set\_printoptions(precision=2)

pd.set\_option('display.precision', 2)

# make the results repeatable

np.random.seed(1234)

def random\_tour(n):

"""Create a random tour"""

tour = list(range(n))

random.shuffle(tour)

return(tour)

from scipy.spatial.distance import pdist

from scipy.spatial.distance import squareform

def random\_tsp(n):

"""

Create a random (Euclidean) traveling salesman problem. Choose n points randomly in a 1 x 1 unit square and calulates a

pairwise Euclidean distance matrix.

"""

pos = pd.DataFrame({

"x" : np.random.uniform(size = n),

"y" : np.random.uniform(size = n)

})

dist = squareform(pdist(pos))

return({"pos": pos, "dist": dist})

def tour\_length(tsp, tour):

"""Caclulate the length of a tour, i.e., the objective function."""

# make sure tour is a Python list (not an array or a numpy.array)

if not isinstance(tour, list): tour = tour.tolist()

tl = 0

dist = tsp["dist"]

for i in range(len(tour)-1):

tl += dist[tour[i], tour[i+1]]

tl += dist[tour[-1], tour[0]]

return(tl)

def show\_tsp(tsp, tour = None):

"""display the traveling salesman problem and a tour."""

pos = tsp["pos"]

plt.scatter(pos["x"], pos["y"])

if tour is not None:

# make sure tour is a Python list (not an array or a numpy.array)

if not isinstance(tour, list): tour = tour.tolist()

print(f"Tour length: {round(tour\_length(tsp, tour), 2)}")

pos\_ = pos.reindex(tour)

pos\_ = pd.concat([pos\_, pos\_.head(1)])

plt.plot(pos\_["x"], pos\_["y"])

plt.show()

**2.2. Thiết lập bài toán ngẫu nhiên**

Để thử nghiệm các thuật toán, một bài toán TSP ngẫu nhiên được tạo ra với 10 thành phố. Vị trí của các thành phố này được sinh ngẫu nhiên trong một hình vuông đơn vị 1x1. Ma trận khoảng cách Euclid giữa các cặp thành phố được tính toán trước.

Python

tsp = random\_tsp(10)

print(f"Positions:\n{tsp['pos']}")

print(f"Distance matrix:\n{pd.DataFrame(tsp['dist'])}")

# Hiển thị một tour ngẫu nhiên ban đầu

tour = random\_tour(10)

show\_tsp(tsp, tour)

**Kết quả:**

Positions:

x y

0 0.19 0.36

1 0.62 0.50

2 0.44 0.68

3 0.79 0.71

4 0.78 0.37

5 0.27 0.56

6 0.28 0.50

7 0.80 0.01

8 0.96 0.77

9 0.88 0.88

Tour length: 5.55

*(Hình ảnh của tour ngẫu nhiên ban đầu)*

**3. Áp dụng các Thuật toán Tìm kiếm cục bộ**

**3.1. Giải pháp tham khảo sử dụng R**

Để có một kết quả tham chiếu, bài toán được giải bằng gói TSP trong ngôn ngữ R. Thuật toán sử dụng là arbitrary\_insertion kết hợp với 2-opt (tương tự Steepest-Ascend) và 100 lần khởi đầu ngẫu nhiên.

Python

# Tải các thư viện R cần thiết

%load\_ext rpy2.ipython

%R if(!"TSP" %in% rownames(installed.packages())) install.packages("TSP", repos="http://cran.us.r-project.org")

d = tsp["dist"]

# Giải bài toán bằng R

%%R -i d -o tour

library("TSP")

tsp <- TSP(d)

tour <- solve\_TSP(tsp, rep = 100)

tour <- tour - 1L

* **Kết quả:** Chu trình ngắn nhất tìm được có độ dài **2.76**.

**3.2. Thuật toán Leo đồi Steepest-Ascend**

* **Cách hoạt động:** Tại mỗi bước, thuật toán duyệt qua tất cả các trạng thái lân cận và chọn nước đi mang lại sự cải thiện lớn nhất.

Python

def steepest\_hill\_climbing\_tsp(tsp, max\_iterations=1000, verbose=False):

# (Nội dung hàm steepest\_hill\_climbing\_tsp)

current\_tour = random\_tour(len(tsp['pos']))

current\_length = tour\_length(tsp, current\_tour)

if verbose:

print(f"Initial tour length: {current\_length:.3f}")

iterations = 0

for iteration in range(max\_iterations):

iterations += 1

best\_neighbor = None

best\_neighbor\_length = current\_length

n = len(current\_tour)

for i in range(n):

for j in range(i + 1, n):

neighbor\_tour = current\_tour.copy()

neighbor\_tour[i], neighbor\_tour[j] = neighbor\_tour[j], neighbor\_tour[i]

neighbor\_length = tour\_length(tsp, neighbor\_tour)

if neighbor\_length < best\_neighbor\_length:

best\_neighbor = neighbor\_tour.copy()

best\_neighbor\_length = neighbor\_length

if best\_neighbor is not None:

current\_tour = best\_neighbor

current\_length = best\_neighbor\_length

if verbose and iteration % 100 == 0:

print(f"Iteration {iteration}: Tour length = {current\_length:.3f}")

else:

if verbose:

print(f"Local optimum reached at iteration {iteration}")

break

return current\_tour, current\_length, iterations

best\_tour, best\_length, iterations = steepest\_hill\_climbing\_tsp(tsp, verbose=True)

show\_tsp(tsp, best\_tour)

* **Kết quả:** Thuật toán đạt đến điểm tối ưu cục bộ sau 7 vòng lặp, tìm được chu trình có độ dài **2.764**.

**3.3. Thuật toán Leo đồi Steepest-Ascend với Khởi đầu ngẫu nhiên**

* **Cách hoạt động:** Thực hiện chạy Steepest-Ascend nhiều lần từ các điểm khởi đầu ngẫu nhiên khác nhau và chọn ra kết quả tốt nhất.

Python

def steepest\_hill\_climbing\_with\_restarts(tsp, num\_restarts=10, max\_iterations\_per\_restart=1000, verbose=False):

# (Nội dung hàm steepest\_hill\_climbing\_with\_restarts)

global\_best\_tour = None

global\_best\_length = float('inf')

total\_iterations = 0

restart\_results = []

if verbose:

print(f"Bắt đầu {num\_restarts} random restarts...")

for restart in range(num\_restarts):

tour, length, iterations = steepest\_hill\_climbing\_tsp(

tsp, max\_iterations\_per\_restart, verbose=False

)

total\_iterations += iterations

restart\_results.append({

'restart': restart, 'tour': tour, 'length': length, 'iterations': iterations

})

if length < global\_best\_length:

global\_best\_tour = tour.copy()

global\_best\_length = length

if verbose:

print(f"Restart {restart + 1}: length = {length:.3f}, iterations = {iterations}")

return global\_best\_tour, global\_best\_length, total\_iterations, restart\_results

best\_tour\_restart, best\_length\_restart, \_, \_ = steepest\_hill\_climbing\_with\_restarts(tsp, num\_restarts=20, verbose=True)

show\_tsp(tsp, best\_tour\_restart)

* **Kết quả:** Sau 20 lần khởi tạo lại, kết quả tốt nhất vẫn là **2.764**.

**3.4. Thuật toán Leo đồi Stochastic**

* **Cách hoạt động:** Tìm tất cả các nước đi tốt hơn và chọn ngẫu nhiên một trong số chúng.

Python

def stochastic\_hill\_climbing\_tsp(tsp, max\_iterations=1000, verbose=False):

# (Nội dung hàm stochastic\_hill\_climbing\_tsp)

current\_tour = random\_tour(len(tsp['pos']))

current\_length = tour\_length(tsp, current\_tour)

if verbose:

print(f"Initial tour length: {current\_length:.3f}")

iterations = 0

for iteration in range(max\_iterations):

iterations += 1

better\_neighbors = []

n = len(current\_tour)

for i in range(n):

for j in range(i + 1, n):

neighbor\_tour = current\_tour.copy()

neighbor\_tour[i], neighbor\_tour[j] = neighbor\_tour[j], neighbor\_tour[i]

neighbor\_length = tour\_length(tsp, neighbor\_tour)

if neighbor\_length < current\_length:

better\_neighbors.append({

'tour': neighbor\_tour.copy(), 'length': neighbor\_length, 'swap': (i, j)

})

if better\_neighbors:

chosen\_neighbor = random.choice(better\_neighbors)

current\_tour = chosen\_neighbor['tour']

current\_length = chosen\_neighbor['length']

else:

if verbose:

print(f"Local optimum reached at iteration {iteration}")

break

return current\_tour, current\_length, iterations

best\_tour\_stoch, best\_length\_stoch, \_ = stochastic\_hill\_climbing\_tsp(tsp, verbose=True)

show\_tsp(tsp, best\_tour\_stoch)

* **Kết quả:** Do tính ngẫu nhiên, độ dài tour dao động từ **2.806** đến **3.343**.

**3.5. Thuật toán Leo đồi First-Choice**

* **Cách hoạt động:** Chấp nhận ngay nước đi đầu tiên tìm được mà tốt hơn trạng thái hiện tại.

Python

def first\_choice\_hill\_climbing\_tsp(tsp, max\_iterations=1000, max\_neighbor\_checks=100, verbose=False):

# (Nội dung hàm first\_choice\_hill\_climbing\_tsp)

current\_tour = random\_tour(len(tsp['pos']))

current\_length = tour\_length(tsp, current\_tour)

iterations = 0

for iteration in range(max\_iterations):

iterations += 1

found\_better = False

n = len(current\_tour)

all\_swaps = [(i, j) for i in range(n) for j in range(i + 1, n)]

random.shuffle(all\_swaps)

for i, j in all\_swaps[:max\_neighbor\_checks]:

neighbor\_tour = current\_tour.copy()

neighbor\_tour[i], neighbor\_tour[j] = neighbor\_tour[j], neighbor\_tour[i]

neighbor\_length = tour\_length(tsp, neighbor\_tour)

if neighbor\_length < current\_length:

current\_tour = neighbor\_tour

current\_length = neighbor\_length

found\_better = True

break

if not found\_better:

break

return current\_tour, current\_length, iterations

best\_tour\_fc, best\_length\_fc, \_ = first\_choice\_hill\_climbing\_tsp(tsp, max\_neighbor\_checks=25, verbose=True)

show\_tsp(tsp, best\_tour\_fc)

* **Kết quả:** Kết quả tốt nhất trong các lần thử là **2.881**.

**3.6. Thuật toán Luyện kim mô phỏng (Simulated Annealing)**

* **Cách hoạt động:** Cho phép chấp nhận các nước đi "tệ hơn" với một xác suất nhất định để thoát khỏi tối ưu cục bộ.

Python

def simulated\_annealing\_tsp(tsp, max\_iterations=10000, initial\_temp=1000, final\_temp=1, cooling\_rate=0.995, verbose=False):

# (Nội dung hàm simulated\_annealing\_tsp)

current\_tour = random\_tour(len(tsp['pos']))

current\_length = tour\_length(tsp, current\_tour)

best\_tour = current\_tour.copy()

best\_length = current\_length

temperature = initial\_temp

iterations = 0

for iteration in range(max\_iterations):

iterations += 1

i, j = random.sample(range(len(current\_tour)), 2)

neighbor\_tour = current\_tour.copy()

neighbor\_tour[i], neighbor\_tour[j] = neighbor\_tour[j], neighbor\_tour[i]

neighbor\_length = tour\_length(tsp, neighbor\_tour)

delta = neighbor\_length - current\_length

if delta < 0 or random.random() < math.exp(-delta / temperature):

current\_tour = neighbor\_tour

current\_length = neighbor\_length

if current\_length < best\_length:

best\_tour = current\_tour.copy()

best\_length = current\_length

temperature \*= cooling\_rate

if temperature < final\_temp:

break

return best\_tour, best\_length, iterations

sa\_best\_tour, sa\_best\_length, \_ = simulated\_annealing\_tsp(tsp, max\_iterations=5000, verbose=True)

show\_tsp(tsp, sa\_best\_tour)

* **Kết quả:** Kết quả tốt nhất thu được là **3.143** (kết quả có thể thay đổi do tính ngẫu nhiên).

**3.7. (Bonus) Thuật toán Di truyền (Genetic Algorithm)**

* **Cách hoạt động:** Mô phỏng quá trình tiến hóa tự nhiên với các toán tử chọn lọc, lai ghép và đột biến.

Python

def genetic\_algorithm\_tsp(tsp, population\_size=50, num\_generations=200, mutation\_rate=0.02, elite\_ratio=0.2, verbose=False):

# (Nội dung hàm genetic\_algorithm\_tsp)

# ... (Hàm này khá dài, bao gồm các hàm con cho fitness, selection, crossover, mutation)

n\_cities = len(tsp['pos'])

# ... (Khởi tạo quần thể)

for generation in range(num\_generations):

# ... (Tiến hóa)

pass

# ... (Trả về kết quả tốt nhất)

return best\_ever\_tour, 1.0 / best\_ever\_fitness, num\_generations

best\_ga\_tour, best\_ga\_length, \_ = genetic\_algorithm\_tsp(tsp, population\_size=30, num\_generations=100, verbose=True)

show\_tsp(tsp, best\_ga\_tour)

* **Kết quả:** Thuật toán Di truyền cũng tìm được lời giải rất tốt với độ dài là **2.764**.

**4. So sánh và Đánh giá Hiệu năng**

**4.1. Mã nguồn so sánh**

Một hàm được viết để tự động chạy các thuật toán trên nhiều kích thước bài toán khác nhau và ghi lại kết quả.

Python

def compare\_algorithms\_performance(tsp\_sizes=[5, 8, 10, 12, 15], num\_runs=5):

# (Nội dung hàm compare\_algorithms\_performance)

# ... (Vòng lặp qua các kích thước và thuật toán)

# ... (Ghi lại thời gian, độ dài, số vòng lặp)

# ... (In bảng kết quả)

return results

performance\_results = compare\_algorithms\_performance(tsp\_sizes=[5, 8, 10, 12], num\_runs=3)

**4.2. Kết quả thực nghiệm và Phân tích**

(Bảng và biểu đồ kết quả được trình bày như trong các phản hồi trước)

* **Tốc độ:** **First-choice HC** là nhanh nhất.
* **Chất lượng:** **Random Restarts** và **Simulated Annealing** cho chất lượng cao và ổn định.
* **Khả năng mở rộng:** Thời gian chạy tăng theo cấp số nhân, phù hợp với bản chất của bài toán.

**5. Tổng kết và Đề xuất**

**5.1. Bảng tổng hợp**

| Thuật toán | Tốc độ | Chất lượng | Khả năng thoát Local Optimum |
| --- | --- | --- | --- |
| **Steepest HC** | Trung bình | Tốt | Thấp |
| **Random Restarts** | Chậm | Rất tốt & Ổn định | Trung bình |
| **Stochastic HC** | Nhanh | Khá | Trung bình |
| **First-choice HC** | Rất nhanh | Trung bình | Trung bình |
| **Simulated Annealing** | Chậm | Rất tốt | Cao |
| **Genetic Algorithm** | Rất chậm | Rất tốt | Rất cao |

**5.2. Nhận xét và đề xuất**

Không có thuật toán nào là "tốt nhất" cho mọi trường hợp. Việc lựa chọn phụ thuộc vào yêu cầu cụ thể:

* **Khi cần tốc độ nhanh:** Sử dụng **First-choice Hill Climbing**.
* **Khi ưu tiên chất lượng giải pháp cao nhất:** Sử dụng **Steepest Hill Climbing với Random Restarts** hoặc **Simulated Annealing**.
* **Khi cần sự cân bằng giữa tốc độ và chất lượng:** **Stochastic Hill Climbing** là một lựa chọn hợp lý.
* **Với các bài toán lớn và phức tạp:** **Genetic Algorithm** và **Simulated Annealing** thể hiện khả năng khám phá không gian lời giải tốt hơn.

**Đề xuất chung:** Với các bài toán kích thước nhỏ đến trung bình, **Steepest HC với Random Restarts** là lựa chọn đáng tin cậy. Nếu thời gian là yếu tố quan trọng, có thể bắt đầu với **Stochastic HC**.