Header

# Folder format

* ElectricCar/
  + evrp-benchmark-set/
  + Statistic/
* The evrp-benchmark-set folder stores all the .evrp file and the Statistic folder holds all the statistic file

# All the function in EVRP

All the code in here are originated from the code in the TR-EVRP-Competition with some modification.

## **For statistic**

def start\_run(*r*: int):

🡺 Hàm này được sử dụng để khởi tạo một lần chạy (run) cho thuật toán heuristic của bạn. Các thao tác được thực hiện bao gồm:

* Thiết lập seed ngẫu nhiên bằng cách sử dụng random.seed(r). Điều này đảm bảo rằng các giá trị ngẫu nhiên sẽ được tạo ra theo cùng một thứ tự cho mỗi lần chạy với cùng một giá trị seed.
* Khởi tạo các giá trị ban đầu cho việc đếm đánh giá (evaluation count) bằng cách gọi init\_evals().
* Khởi tạo giá trị tốt nhất hiện tại cho lần chạy bằng cách gọi init\_current\_best().
* In ra thông báo cho biết lần chạy hiện tại và giá trị seed ngẫu nhiên.

def end\_run(*r*: int):

🡺 Hàm này được sử dụng để kết thúc một lần chạy của thuật toán heuristic. Các thao tác được thực hiện bao gồm:

* Tính giá trị trung bình của các lần chạy trước đó và giá trị tốt nhất hiện tại bằng cách gọi get\_mean(r - 1, get\_current\_best()).
* In ra thông báo cho biết lần chạy đã kết thúc, giá trị chất lượng tốt nhất và tổng số đánh giá đã thực hiện.

def termination\_condition():

🡺 Hàm này được sử dụng để đặt điều kiện kết thúc cho thuật toán heuristic của bạn. Hàm kiểm tra xem điều kiện kết thúc đã được đáp ứng hay chưa.

Nếu số lần đánh giá đã đạt đến hoặc vượt qua mức kết thúc (được định nghĩa bởi TERMINATION), hàm sẽ trả về True để chỉ ra rằng thuật toán nên kết thúc, ngược lại sẽ trả về False.

def open\_stats()

🡺 Hàm này được sử dụng để khởi tạo và mở các tệp xuất phát để lưu trữ thống kê hiệu suất. Các thao tác bao gồm:

* Khởi tạo danh sách perf\_of\_trials với các giá trị ban đầu là 0.0.
* Lấy tên vấn đề từ biến problem\_instance và định dạng chuỗi perf\_filename để đại diện cho tên tệp xuất phát.
* Mở tệp xuất phát để ghi các thống kê hiệu suất, và gán đối tượng tệp vào biến log\_performance.
* Nếu không thể mở tệp, in ra thông báo lỗi và thoát khỏi chương trình.

def get\_mean(*r*: int, *value*: float):

🡺 Hàm này được sử dụng để cập nhật giá trị thể hiện chất lượng giải pháp của mỗi lần chạy vào danh sách perf\_of\_trials. Các tham số là chỉ số r của lần chạy và giá trị value của chất lượng giải pháp.

def mean(*values*, *size*: int):

🡺 Hàm này được sử dụng để cập nhật giá trị thể hiện chất lượng giải pháp của mỗi lần chạy vào danh sách perf\_of\_trials. Các tham số là chỉ số r của lần chạy và giá trị value của chất lượng giải pháp.

def stdev(*values*, *size*: int, *average*: float):

🡺 Hàm này tính độ lệch chuẩn của một danh sách giá trị values có kích thước size và giá trị trung bình average. Nó tính tổng các bình phương của hiệu giữa mỗi giá trị và giá trị trung bình, sau đó chia cho kích thước - 1 và lấy căn bậc hai để tìm độ lệch chuẩn và trả về giá trị đó.

def best\_of\_vector(*values*, *l*: int):

🡺 Hàm này tìm giá trị nhỏ nhất trong danh sách values có kích thước l. Nó so sánh từng giá trị với giá trị nhỏ nhất hiện tại và cập nhật nếu tìm thấy giá trị nhỏ hơn. Sau đó, nó trả về giá trị nhỏ nhất đó.

def worst\_of\_vector(*values*, *l*: int):

🡺 Hàm này tìm giá trị lớn nhất trong danh sách values có kích thước l. Nó so sánh từng giá trị với giá trị lớn nhất hiện tại và cập nhật nếu tìm thấy giá trị lớn hơn. Sau đó, nó trả về giá trị lớn nhất đó.

def close\_stats()

🡺 Hàm này được sử dụng để đóng tệp xuất phát và ghi các thống kê hiệu suất đã tính toán. Các thao tác bao gồm:

* Ghi các giá trị của perf\_of\_trials thành dòng riêng lẻ trong tệp xuất phát.
* Tính giá trị trung bình, độ lệch chuẩn, giá trị nhỏ nhất và giá trị lớn nhất của perf\_of\_trials.
* Ghi các giá trị thống kê vào tệp xuất phát.
* Đóng tệp xuất phát.

## **For EVRP**

def euclidean\_distance(*i*: int, *j*: int):

🡺 Tính toán và trả về khoảng cách Euclid giữa hai đối tượng.

def compute\_distances():

🡺 Tính ma trận khoảng cách của bài toán.

def generate\_2D\_matrix\_double(*n*: int, *m*: int):

🡺 Tạo và trả về một mảng hai chiều kiểu double.

def read\_problem(*filename*: str):

🡺 Đọc bài toán và tạo vectơ đối tượng ban đầu.

def fitness\_evaluation(*routes*, *size*: int):

🡺 Đánh giá chất lượng của một giải pháp. Nó nhận đầu vào là một mảng các chỉ số nút (routes) và độ dài của mảng (size).

def print\_solution(*routes*, *size*: int):

🡺 In ra giải pháp dưới dạng các routes. Nó nhận đầu vào là một mảng các chỉ số nút (routes) và độ dài của mảng (size).

def get\_distance(*\_from*: int, *\_to*: int):

🡺 Trả về khoảng cách giữa hai nút được chỉ định.

def get\_energy\_consumption(*\_from*: int, *\_to*: int):

🡺 Hàm "get\_energy\_consumption" trong đoạn mã trên có tác dụng trả về lượng năng lượng tiêu thụ khi di chuyển giữa hai điểm: từ điểm "\_from" đến điểm "\_to".

def get\_customer\_demand(*customer*: int):

🡺 Trả về yêu cầu (demand) cho một khách hàng cụ thể. Nó nhận đầu vào là chỉ số của khách hàng (customer).

def is\_charging\_station(*node*: int):

🡺 Trả về giá trị true nếu một nút cụ thể là trạm sạc (charging station), và trả về giá trị false trong trường hợp khác.

def get\_current\_best():

🡺 trả về chất lượng của giải pháp tốt nhất cho đến thời điểm hiện tại.

def get\_current\_best\_route():

🡺 trả về đường đi của giải pháp tốt nhất cho đến thời điểm hiện tại.

def init\_current\_best():

🡺 Được sử dụng để khởi tạo lại các biến "current\_best" và "best\_routes" để bắt đầu một lần chạy độc lập mới, nơi ta muốn thiết lập các giá trị ban đầu cho giải pháp tốt nhất.

def get\_evals():

🡺 Sử dụng biến toàn cục evals để truy cập và trả về giá trị hiện tại của số lượng đánh giá đã được thực hiện (fitness evaluations).

def init\_evals():

🡺 Đặt lại giá trị của biến toàn cục "evals" về 0.0 để bắt đầu một lần chạy độc lập mới.

def check\_solution(*t*, *size*: int):

🡺 Kiểm tra tính hợp lệ của các đường đi trong giải pháp bằng cách xác minh các ràng buộc về sức chứa, năng lượng và các yêu cầu khách hàng. Nếu giải pháp không đáp ứng các ràng buộc này, chương trình sẽ kết thúc và in thông báo lỗi.

## **For heuristic**

### **Metaheuristic**

* Phần này chứa tất cả các hàm để chạy thuật toán heuristic.
* Chúng ta sử dụng meta-heuristic để giải quyết bài toán EVRP (Electric Vehicle Routing Problem).
* Meta-heuristic có hai đặc điểm: tăng cường (intensification) và đa dạng hóa (diversification).
  + Đối với dạng hóa, chúng ta sử dụng Variable Neighborhood Search (VNS - Tìm kiếm Điểm kề Biến đổi).
    - Trong giai đoạn xáo trộn (perturbation), chúng ta ngẫu nhiên sử dụng các phép biến đổi như xáo trộn (shuffle), đảo ngược (invert) và kết nối lại (reconnect) để tạo ra một đường đi mới, sau đó sử dụng thuật toán SSF để khắc phục lỗi trong đường đi.
  + Đối với giai đoạn tăng cường (intensification), chúng ta sử dụng Variable Neighborhood Descent (VND) hoặc Randomized Variable Neighborhood Descent (RVND) làm quy trình tìm kiếm cục bộ.
    - Các toán tử được sử dụng trong quy trình tìm kiếm cục bộ bao gồm: trao đổi (exchange), tái đặt vị trí (relocation), hai-opt (two-opt), or-opt và AFS reallocation.

def initialize\_heuristic():

🡺 Hàm này không có nội dung, chỉ đơn giản là để khởi tạo heuristic.

def run\_heuristic():

🡺 Hàm này thực hiện chạy thuật toán heuristic.

* Đầu tiên, nó xây dựng một giải pháp ban đầu bằng cách gọi hàm initial\_construction().
* Sau đó, trong mỗi vòng lặp, giải pháp hiện tại được biến đổi (perturbation) để tạo ra một đường đi mới bằng cách gọi hàm perturbation().
* Tiếp theo, đường đi mới được cải thiện bằng cách áp dụng quá trình tìm kiếm cục bộ (local search) thông qua hàm local\_search().
* Nếu đường đi mới có độ thích nghi tốt hơn so với đường đi hiện tại, thì đường đi hiện tại được cập nhật. Quá trình lặp tiếp tục cho đến khi điều kiện dừng được đáp ứng.

def local\_search(*solution\_route*: list):

🡺 Hàm này thực hiện quá trình tìm kiếm cục bộ trên một giải pháp cụ thể (solution\_route).

* Trong quá trình tìm kiếm, các phương pháp tìm kiếm cục bộ (như relocate(), exchange(), two\_opt(), or\_opt(), AFS\_reallocation()) được chọn ngẫu nhiên hoặc tuần tự để cố gắng cải thiện giải pháp.
* Nếu một phương pháp cải thiện giải pháp, thì giải pháp hiện tại được cập nhật và quá trình tìm kiếm bắt đầu lại từ đầu.
* Hàm này trả về giải pháp cải thiện và một biến dừng để kiểm tra xem có tiếp tục quá trình tìm kiếm hay không.

def initial\_construction():

🡺 Hàm này thực hiện quá trình xây dựng ban đầu của giải pháp.

* Phương pháp xây dựng ban đầu có thể được chọn từ các phương pháp khác nhau như nearest\_neighbour(), clarke\_wright\_saving(), minimum\_spanning\_tree().
* Sau đó, kết quả của quá trình xây dựng ban đầu được cải thiện bằng cách gọi hàm seperate\_sequential\_fixing().

def perturbation(*solution\_route*: list):

🡺 Hàm này thực hiện quá trình biến đổi (perturbation) trên một giải pháp cụ thể (solution\_route).

* Quá trình biến đổi bao gồm: chia giải pháp thành các phần con, xáo trộn các phần con và đảo ngược các phần con.
* Kết quả của quá trình biến đổi là một giải pháp mới. Hàm này trả về giải pháp mới đã được biến đổi.

def remove\_duplicate\_node(*solution\_route*: list):

🡺 Hàm này loại bỏ các nút trùng lặp trong một giải pháp. Nếu có các nút liên tiếp giống nhau, nút thứ hai sẽ bị xóa. Hàm này trả về giải pháp đã loại bỏ các nút trùng lặp.

### **Initial construction**

Mỗi phương pháp dưới đây sẽ tạo ra một đường đi đi qua tất cả các khách hàng, bất kể ràng buộc về khả năng chứa và mức pin. Nó bắt đầu từ điểm gốc (depot) và kết thúc tại điểm gốc (depot).

def nearest\_neighbour():

* Hàm này tạo ra một tuyến đường duy nhất qua tất cả các điểm khách hàng (các thành phố) sao cho khoảng cách tổng là ngắn nhất có thể.
* Hàm trả về một danh sách (nn\_solution) chứa các điểm đã được sắp xếp theo thứ tự của tuyến đường.

def clarke\_wright\_saving():

* Giải thuật này tập trung vào việc tìm cách kết hợp các tuyến đường để giảm chi phí.
* Hàm này tạo ra một tuyến đường duy nhất qua tất cả các điểm khách hàng (các thành phố) sao cho tổng chi phí là thấp nhất có thể.
* Hàm trả về một danh sách (route) chứa các điểm đã được sắp xếp theo thứ tự của tuyến đường.

class Graph:

🡺 Lớp Graph đại diện cho một đồ thị không hướng với các phương thức để thêm cạnh và thực hiện các thao tác liên quan đến MST (Minimum Spanning Tree) sử dụng thuật toán Kruskal.

* def \_\_init\_\_(*self*, *vertices*):Phương thức khởi tạo lớp Graph với số đỉnh vertices và một đồ thị rỗng được lưu trữ trong thuộc tính graph.
* def addEdge(*self*, *u*, *v*, *w*):Phương thức này được sử dụng để thêm một cạnh vào đồ thị. Điểm đầu của cạnh là u, điểm cuối là v, và trọng số là w. Cạnh được thêm vào danh sách graph của đối tượng Graph.
* def find(*self*, *parent*, *i*): Một phương thức hỗ trợ để tìm đồng tố của một phần tử i trong tập hợp các phần tử parent. Đây là một phương thức sử dụng kỹ thuật nén đường dẫn (path compression).
* def union(*self*, *parent*, *rank*, *x*, *y*): Một phương thức hỗ trợ để gộp hai tập hợp x và y vào cùng một tập hợp lớn hơn. Phương pháp này sử dụng kỹ thuật gộp theo rank (union by rank) để duy trì cân bằng cây.
* def KruskalMST(*self*): Phương thức chính để xây dựng MST bằng cách sử dụng thuật toán Kruskal. Phương thức này trả về một tuyến đường (route) qua tất cả các điểm trong đồ thị.

def minimum\_spanning\_tree():

Hàm này thực hiện tìm cây bao trùm tối thiểu của đồ thị sử dụng phương pháp Kruskal. Nó sử dụng một đối tượng Graph và ma trận d để khởi tạo đồ thị và tính toán khoảng cách giữa các đỉnh. Sau đó, nó gọi phương thức KruskalMST để lấy cây bao trùm tối thiểu và trả về một tuyến đường (route) qua các đỉnh của cây bao trùm tối thiểu.

### **Repair method**

def seperate\_sequential\_fixing(*solution\_route*: list):

Hàm này thực hiện quy trình sửa chữa tuần tự trên solution\_route đầu vào. Hàm sửa đổi solution\_route bằng cách chèn các nút trạm và nút trạm pin để đảm bảo ràng buộc về tải và pin được đáp ứng.

def get\_nearest\_station(*node*: int):

Hàm này nhận đầu vào là một nút và trả về chỉ mục của nút trạm gần nhất.

def get\_current\_battery\_consumption(*solution\_route*: list, *index*: int):

Hàm này tính toán mức pin hiện tại từ đầu đến chỉ mục hiện tại trong solution\_route.

### **Local search operator**

* Là một phép toán cụ thể được sử dụng để khám phá và cải thiện chất lượng của một giải pháp trong một vùng lân cận cục bộ.
* Các local search operator được thiết kế để thực hiện các thay đổi như hoán đổi (swap), chèn (insertion), xoá (deletion), đảo ngược (reversal) hoặc kết hợp các thay đổi này để tạo ra các giải pháp mới. Các operator này thường được áp dụng lặp đi lặp lại trong quá trình tìm kiếm cục bộ để tìm kiếm các giải pháp tiềm năng và tiến dần đến một giải pháp tốt nhất trong vùng lân cận cục bộ.

def two\_opt(*solution\_route*: list, *steps*: int):

def cost\_two\_opt(*id1*: int, *id2*: int, *id3*: int, *id4*: int):

🡺 Hàm two\_opt và cost\_two\_opt được sử dụng để thực hiện phép biến đổi "2-opt" trên một tuyến đường hiện tại để tìm kiếm một tuyến đường mới có chiều dài tổng quát tốt hơn.

def exchange(*solution\_route*: list, *steps*: int):

def cost\_exchange(*id1*: int, *id2*: int, *id3*: int, *id4*: int, *id5*: int, *id6*: int):

🡺 Hàm exchange và cost\_exchange được sử dụng để thực hiện phép trao đổi vị trí giữa hai đoạn trong tuyến đường hiện tại để tìm kiếm một tuyến đường mới có chi phí tổng quát tốt hơn.

def relocate(*solution\_route*: list, *steps*: int):

def cost\_relocate(*id1*: int, *id2*: int, *id3*: int, *id4*: int, *id5*: int):

🡺 Hàm relocate được sử dụng để tạo ra một tuyến đường mới bằng cách di chuyển một đỉnh từ một vị trí trong tuyến đường hiện tại đến một vị trí khác để cải thiện chiều dài tổng quát của tuyến đường.

def or\_opt(*solution\_route*: list, *steps*: int):

def cost\_or\_opt(*id1*: int, *id2\_first*: int, *id3\_first*: int, *id4*: int, *id5*: int, *id6\_second*: int, *id7\_second*: int, *id8*: int):

🡺 Hàm or\_opt được sử dụng để tạo ra một tuyến đường mới bằng cách thay thế hoặc chèn các đoạn từ hai vị trí liên tiếp trong tuyến đường hiện tại vào hai vị trí khác nhau trong tuyến đường để cải thiện chiều dài tổng quát của tuyến đường.

🡺 Hàm cost\_or\_opt là một hàm hỗ trợ được sử dụng trong or\_opt. Nó tính toán giá trị chi phí của phép biến đổi thay thế hoặc chèn các đoạn từ vị trí i đến i+1 và từ vị trí j đến j+1 trong tuyến đường. Cụ thể, nó tính toán các khoảng cách giữa các điểm trong các đoạn được thay đổi và trả về hiệu của tổng các khoảng cách ban đầu và tổng các khoảng cách sau khi thay đổi.

def AFS\_reallocation(*solution\_route*: list, *steps*: int):

🡺 Hàm AFS\_reallocation được sử dụng để thực hiện việc tái phân bổ vị trí trong tuyến đường của xe. Quá trình tái phân bổ này bao gồm loại bỏ các vị trí trùng lặp và trạm sạc, và tách các đoạn liên tiếp để tối ưu hóa việc sử dụng pin.

- Kết quả cuối cùng là tuyến đường đã được tái phân bổ và được trả về bởi hàm.

def check\_solution\_heuristic(*route*: list):

🡺 Hàm được sử dụng để kiểm tra tính hợp lệ của một tuyến đường (route) dựa trên các tiêu chí heuristic (nguyên tắc) nhất định. Hàm này kiểm tra xem tuyến đường có tuân thủ các ràng buộc về năng lượng và sức chứa hay không.

### **For drawing graph**

def draw\_graph(*solution\_route*, *node\_list*, *cust\_demand*, *charging\_station*, *ACTUAL\_PROBLEM\_SIZE*, *run*: int):

🡺 Hàm này được sử dụng để vẽ đồ thị biểu diễn tuyến đường và các điểm trong bài toán.

Quá trình vẽ đồ thị bao gồm các bước sau:

1. Khởi tạo một hình tròn đại diện cho điểm xuất phát (depot) với tọa độ và màu sắc tương ứng.
2. Tạo danh sách các hình vuông đại diện cho trạm sạc (charging station).
3. Tạo danh sách các hình tròn đại diện cho khách hàng (customer).
4. Thêm các hình tròn và hình vuông vào đồ thị.
5. Vẽ các đoạn thẳng màu xanh lá cây đại diện cho từng phân đoạn của tuyến đường.
6. Cài đặt thuộc tính cho trục x và trục y.
7. Hiển thị đồ thị.

# Main code

# Run code