Documentação Detalhada dos Scripts de Simulação SUMO

Introdução

Esta documentação fornece uma análise detalhada de um conjunto de scripts Python projetados para trabalhar com o simulador de tráfego SUMO. O objetivo principal do sistema é simular e analisar o comportamento de veículos elétricos em um ambiente urbano, com foco na alocação otimizada de postos de recarga (Parking Areas - PAs).

O sistema é composto por três scripts principais:

1. gerar_rotas_sem_vtype.py: Responsável por gerar um arquivo de rotas inicial para a simulação.
2. definir_eletricos.py: Modifica o arquivo de rotas para definir uma porcentagem de veículos como elétricos.
3. controlador_pa_opt.py: O script principal que executa a simulação, controla o comportamento dos veículos e implementa diferentes heurísticas para a alocação de postos de recarga.

A seguir, cada script será documentado em detalhes, explicando sua funcionalidade, parâmetros, e como ele se integra ao fluxo de trabalho geral da simulação.

1. gerar_rotas_sem_vtype.py

1.1. Objetivo

O script gerar_rotas_sem_vtype.py é o ponto de partida para a criação de um cenário de simulação no SUMO. Sua função é gerar um arquivo de rotas (.rou.xml) contendo um número especificado de viagens aleatórias dentro de uma rede de tráfego pré-definida (neste caso, cologne2.net.xml).

Este script utiliza a ferramenta randomTrips.py, que é parte do conjunto de ferramentas do SUMO, para criar as viagens. É importante notar que, nesta etapa, os veículos ainda não têm um tipo (vType) definido, o que significa que não são diferenciados como veículos a combustão ou elétricos.

1.2. Funcionalidades Principais

- **Geração de Rotas Aleatórias**: Utiliza o randomTrips.py do SUMO para criar um arquivo de rotas com um número definido de viagens.
- **Interface de Linha de Comando**: Permite que o usuário especifique o número total de viagens a serem geradas através de um argumento de linha de comando.

- **Simplicidade**: Foca em uma única tarefa, gerando um arquivo de rotas base que será modificado posteriormente.

1.3. Parâmetros de Linha de Comando

- -t ou --total-trips: (Opcional) Um número inteiro que define o número total de viagens a serem geradas. O valor padrão é 10000.

1.4. Exemplo de Uso

Para gerar um arquivo de rotas com 15.000 viagens, o seguinte comando pode ser usado:

```
python gerar_rotas_sem_vtype.py --total-trips 15000
```

Isso criará um arquivo chamado rotas.rou.xml no mesmo diretório.

1.5. Código Fonte Comentado

```
import os
```

```
import subprocess
import argparse

def gerar_rotas_sem_vtype(output_file, num_trips):
    """
    Gera um arquivo de rotas com um número total de viagens, sem
especificar vTypes.

    Args:
        output_file (str): O nome do arquivo de rotas de saída (ex:
"rotas.rou.xml").
        num_trips (int): O número total de viagens a serem geradas.
    """
    # Comando para executar a ferramenta randomTrips.py do SUMO
    comando = [
        "python",
        # Caminho para o script randomTrips.py.
        # ATENÇÃO: Este caminho pode precisar ser ajustado dependendo da
sua instalação do SUMO.
        "/home/alex/sumo/sumo-1.18.0/tools/randomTrips.py",
        "-n", "cologne2.net.xml",  # Arquivo de rede do SUMO
        "-r", output_file,         # Nome do arquivo de rotas de saída
        "-e", str(num_trips),       # Número de viagens a serem geradas
        "-p", "1.0",                # Período de tempo para a geração das
viagens
        "--vehicle-class", "passenger", # Classe do veículo
```

```
        "--prefix", ""                # Prefixo para os IDs dos veículos
    ]
    # Executa o comando e verifica se houve erros
    subprocess.run(comando, check=True)
    print(f"Arquivo de rotas \'{output_file}\' gerado (sem vTypes).")

if __name__ == "__main__":
    # Configura o parser de argumentos de linha de comando
    parser = argparse.ArgumentParser(description="Gera um arquivo de
rotas para a simulação SUMO sem especificar vTypes.")
    parser.add_argument("-t", "--total-trips", type=int, default=10000,
help="Número total de viagens a serem geradas.")
    args = parser.parse_args()

    # Nome do arquivo de saída padrão
    arquivo_saida = "rotas.rou.xml"
    # Chama a função para gerar as rotas
    gerar_rotas_sem_vtype(arquivo_saida, args.total_trips)
```

2. definir_eletricos.py

2.1. Objetivo

Após a geração do arquivo de rotas base, o script definir_eletricos.py entra em ação para diferenciar os veículos. Sua principal responsabilidade é modificar o arquivo de rotas (.rou.xml) para definir uma porcentagem específica de veículos como elétricos.

Este script também adiciona as definições de tipo de veículo (vType) para "veículo normal" e "veículo elétrico" no arquivo de rotas, se elas ainda não existirem. A definição do veículo elétrico inclui parâmetros detalhados da bateria, como capacidade, eficiência, e outros, baseados no modelo soulEV65.

2.2. Funcionalidades Principais

- **Definição de Tipos de Veículo**: Adiciona vType para veículos normais e elétricos ao arquivo de rotas.
- **Seleção Aleatória**: Seleciona aleatoriamente uma porcentagem de veículos do arquivo de rotas para serem designados como elétricos.
- **Modificação de Arquivo XML**: Lê o arquivo de rotas de entrada, modifica os atributos dos veículos e salva o resultado em um novo arquivo.
- **Geração em Lote**: O script é projetado para gerar múltiplos arquivos de rotas com diferentes porcentagens de veículos elétricos, facilitando a execução de simulações com cenários variados.

2.3. Parâmetros

O script, em seu if __name__ == "__main__":, está configurado para gerar arquivos de rotas para diferentes porcentagens de veículos elétricos (5%, 10%, 20%) e criar 5 arquivos para cada porcentagem. Esses valores podem ser facilmente modificados no código.

2.4. Exemplo de Uso

O script é executado diretamente, sem a necessidade de argumentos de linha de comando:

```
python definir_eletricos.py
```

Isso irá gerar arquivos como rotas_5_0_mod.rou.xml, rotas_10_0_mod.rou.xml, etc., no mesmo diretório.

2.5. Código Fonte Comentado

```
import xml.etree.ElementTree as ET
```

```
import random

def definir_eletricos(input_file, output_file, electric_percentage):
    """
    Modifica um arquivo de rotas para definir uma porcentagem de viagens
como elétricas,
    selecionando-as aleatoriamente.

    Args:
        input_file (str): O nome do arquivo de rotas de entrada.
        output_file (str): O nome do arquivo de rotas de saída
modificado.
        electric_percentage (float): A porcentagem de veículos a serem
definidos como elétricos (0.0 a 1.0).
    """
    tree = ET.parse(input_file)
    root = tree.getroot()

    # Adiciona a definição de vType para veículo normal, se não existir
    if not root.find('vType[@id="veiculo_normal"]'):
        vtype_normal = ET.Element("vType", attrib={...}) # Atributos
omitidos para brevidade
        root.insert(0, vtype_normal)

    # Adiciona a definição de vType para veículo elétrico, se não existir
    if not root.find('vType[@id="electric_vehicle"]'):
        vtype_eletrico = ET.Element("vType", attrib={...}) # Atributos
omitidos para brevidade
```

```
        # Adiciona parâmetros da bateria
        vtype_eletrico.append(ET.Element("param", attrib={"key":
"has.battery.device", "value": "true"}))
        # ... outros parâmetros da bateria ...
        root.insert(1, vtype_eletrico)

    # Coleta todos os veículos do arquivo
    all_vehicles = root.findall('vehicle')

    # Calcula o número de veículos elétricos com base na porcentagem
    num_vehicles = len(all_vehicles)
    num_electric = int(num_vehicles * electric_percentage)

    # Seleciona aleatoriamente os veículos que serão elétricos
    electric_vehicles = random.sample(all_vehicles, num_electric)

    # Define o tipo dos veículos selecionados como 'electric_vehicle'
    for vehicle in electric_vehicles:
        vehicle.set("type", "electric_vehicle")

    # Define o tipo dos veículos restantes como 'veiculo_normal'
    for vehicle in all_vehicles:
        if vehicle not in electric_vehicles:
            vehicle.set("type", "veiculo_normal")

    # Salva o arquivo de rotas modificado
    tree.write(output_file)

if __name__ == "__main__":
    # Define as porcentagens de veículos elétricos a serem testadas
    porcentagens = [0.05, 0.10, 0.20]
    # Gera 5 arquivos de rotas para cada porcentagem
    for i in range(5):
        for porcentagem in porcentagens:
            input_file = "rotas.rou.xml"
            output_file = f"rotas_{porcentagem *
100:.0f}_{i}_mod.rou.xml"
            definir_eletricos(input_file, output_file, porcentagem)
```

```
            print(f"Arquivo '{output_file}' gerado com {porcentagem *
100:.0f}% de carros elétricos.")
```

3. controlador_pa_opt.py

3.1. Objetivo

O controlador_pa_opt.py é o script central e mais complexo do sistema de simulação. Ele orquestra a execução da simulação no SUMO, implementa diferentes heurísticas para a

alocação de postos de recarga (Parking Areas - PAs) para veículos elétricos, e coleta dados para análise de desempenho.

O script opera em dois modos principais:

- **first_run**: Executa uma simulação inicial para coletar dados sobre as rotas mais utilizadas pelos veículos elétricos. Com base nesses dados, ele seleciona as localizações para os postos de recarga usando uma das heurísticas disponíveis (Random, Greedy, ou GRASP).
- **second_run**: Executa uma segunda simulação, desta vez com os postos de recarga já alocados, para avaliar o desempenho da solução encontrada. Métricas como tempo de espera na fila, distância até a estação de recarga, e número de teletransportes (indicativo de problemas na simulação) são coletadas.

3.2. Funcionalidades Principais

- **Controle da Simulação SUMO**: Inicia e controla a simulação SUMO usando a biblioteca TraCI (Traffic Control Interface).
- **Coleta de Dados**: Durante a simulação, coleta dados sobre as lanes (faixas) mais visitadas pelos veículos elétricos.
- **Heurísticas de Alocação de PAs**:
  - **Random**: Seleciona aleatoriamente as localizações para os postos de recarga a partir das lanes visitadas.
  - **Greedy**: Seleciona as lanes mais visitadas como localizações para os postos de recarga.
  - **GRASP (Greedy Randomized Adaptive Search Procedure)**: Uma heurística mais sofisticada que tenta encontrar uma solução de boa qualidade, balanceando a popularidade das lanes com a dispersão geográfica dos postos de recarga.
- **Geração de Arquivos de Configuração**: Gera os arquivos .add.xml que definem as localizações dos postos de recarga para a simulação no SUMO.
- **Análise de Resultados**: Coleta e salva os resultados da simulação em um arquivo CSV para análise posterior.
- **Otimizações de Desempenho**: Utiliza técnicas como pré-indexação da rede e cache de funções (lru_cache) para acelerar o cálculo de distâncias e a execução da heurística GRASP.

3.3. Parâmetros de Linha de Comando

O script aceita uma variedade de argumentos de linha de comando para configurar a simulação:

- --route_file: (Obrigatório) O arquivo de rotas a ser usado na simulação.
- --method: (Obrigatório) A heurística a ser usada para a alocação de PAs (random, greedy, ou grasp).
- --mode: (Obrigatório) O modo de execução (first_run ou second_run).

- --add_file: (Opcional) O arquivo .add.xml com as localizações dos PAs (obrigatório para o second_run).
- --net_file: (Opcional) O arquivo de rede do SUMO. Padrão: cologne2.net.xml.
- --er: (Opcional) O número de estações de recarga a serem alocadas. Padrão: 10.
- --tr_min: (Opcional) O tempo de recarga em minutos. Padrão: 10.
- --capacity: (Opcional) O número de vagas por estação de recarga. Padrão: 5.
- --seed: (Opcional) A semente para os geradores de números aleatórios. Padrão: 42.
- --rep: (Opcional) O número da repetição (usado para identificar os resultados no arquivo CSV). Padrão: 1.
- --threads: (Opcional) O número de threads a serem usadas pelo SUMO. Padrão: 24.
- --step_length: (Opcional) O tamanho do passo da simulação em segundos. Padrão: 1.5.
- --out_dir: (Opcional) O diretório de saída para os arquivos de resultados. Padrão: output.

3.4. Fluxo de Execução

4 **first_run**:
   4.1 Executa a simulação com um arquivo de rotas modificado (com veículos elétricos definidos).
   4.2 Coleta as lanes visitadas pelos veículos elétricos e a frequência de visitas.
   4.3 Salva os dados de visitas em um arquivo .pkl.
   4.4 Aplica a heurística selecionada (random, greedy, ou grasp) para escolher as lanes onde os PAs serão alocados.
   4.5 Gera um arquivo .add.xml contendo a definição dos PAs.

5 **second_run**:
   5.1 Executa a simulação novamente, mas desta vez incluindo o arquivo .add.xml com os PAs.
   5.2 Durante a simulação, monitora o comportamento dos veículos elétricos, especialmente aqueles com bateria baixa.
   5.3 Coleta métricas de desempenho, como tempo de espera na fila para recarga e distância percorrida até a estação.
   5.4 Salva os resultados em um arquivo resultados_execucoes.csv.

3.5. Código Fonte Comentado (Seções Relevantes)

Devido à sua extensão, o código completo não será reproduzido aqui. Em vez disso, as seções mais importantes serão destacadas e explicadas.

3.5.1. Pré-indexação e Cache

Para otimizar o desempenho, o script pré-indexa a rede do SUMO no início da execução. Isso evita a necessidade de analisar o arquivo XML da rede repetidamente. Além disso, a função

compute_lane_proximity_cached utiliza o decorador @lru_cache para armazenar os resultados de cálculos de distância, o que acelera significativamente a heurística GRASP.

```
# --- ADICIONADO (S1): índices globais e grafo para a função cacheada

_graph = None
_lane_to_edge = {}
_edge_nodes = {}
_lane_len_index = {}

# --- ADICIONADO (S1): Pré-indexa a rede UMA vez
def build_net_indexes(net_file):
    # ... (código para ler o XML da rede e preencher os dicionários de
índice)
    return lane_to_edge, edge_nodes, lane_len

# --- ADICIONADO (S2): Proximidade cacheada sem reparse de XML
@lru_cache(maxsize=None)
def compute_lane_proximity_cached(lane1, lane2, distance_threshold=None):

    # ... (código para calcular a distância entre duas lanes usando os
índices pré-computados)
```

3.5.2. Heurística GRASP

A implementação do GRASP é um dos pontos centrais do script. Ele tenta construir uma solução de alta qualidade de forma iterativa, combinando elementos de aleatoriedade e um critério guloso.

```
def select_grasp_stations(lane_visits, num_stations, graph, net_file):

    print("----- Método: GRASP (Ultra Otimizado) -----")
    best_solution = []
    best_score = -1
    visited_lanes = list(lane_visits.keys())

    # Cria uma Lista de Candidatos Restrita (LRC) com as lanes mais
visitadas
    lrc = sorted(visited_lanes, key=lambda lane: lane_visits.get(lane,
0), reverse=True)[:num_stations * 5]

    # Executa várias iterações para encontrar a melhor solução
    for _ in range(10):
        current_solution = []
        tabu_set = set()

        # ... (lógica para construir uma solução iterativamente,
```

```
        #      selecionando lanes da LRC e evitando lanes muito próximas)

        # Avalia a solução atual
        current_score = sum(lane_visits[lane] for lane in
current_solution if lane in lane_visits)

        # Atualiza a melhor solução encontrada
        if current_score > best_score:
            best_score = current_score
            best_solution = current_solution

    print("Lanes selecionadas (GRASP Ultra Otimizado):", best_solution)
```

```
    return best_solution
```

3.5.3. Execução da Simulação

A função run_simulation é responsável por iniciar o SUMO com os parâmetros corretos e controlar o loop da simulação. Dentro do loop, ela executa a lógica específica para cada modo (first_run ou second_run), como coletar visitas às lanes ou monitorar o estado da bateria dos veículos.

```
def run_simulation(args, graph, add_file=None):
```

```
    # ... (configuração do comando SUMO)

    traci.start(sumoCmd)

    lane_visits = {}
    low_battery_vehicles = set_low_battery_percentage(args.route_file,
LOW_BATTERY_PERCENTAGE)
    visited_parking = {}

    while traci.simulation.getMinExpectedNumber() > 0:
        traci.simulationStep()

        if args.mode == "first_run":
            # Coleta as lanes visitadas pelos veículos elétricos
            # ...
        elif args.mode == "second_run":
            # Monitora a bateria dos veículos e o uso dos PAs
            # ...

    traci.close()
```

```
    # ... (cálculo e retorno dos resultados)
```

4. Visão Geral do Sistema e Fluxo de Trabalho

O sistema de simulação é projetado para operar em um fluxo de trabalho sequencial, onde a saída de um script serve como entrada para o próximo. O diagrama a seguir ilustra o fluxo de trabalho completo:

```
graph TD
```

```
    A[Início] --> B{gerar_rotas_sem_vtype.py};
    B --> C[rotas.rou.xml];
    C --> D{definir_eletricos.py};
    D --> E[rotas_mod.rou.xml];
    E --> F{controlador_pa_opt.py (first_run)};
    F --> G[lane_visits.pkl];
    F --> H[parking_areas.add.xml];
    E --> I{controlador_pa_opt.py (second_run)};
    H --> I;
    I --> J[resultados_execucoes.csv];
```

```
    J --> K[Análise de Resultados];
```

Passos do Fluxo de Trabalho:

6 **Geração de Rotas Base**: O processo começa com a execução de gerar_rotas_sem_vtype.py para criar um arquivo de rotas (rotas.rou.xml) com um grande número de viagens aleatórias.

7 **Definição de Veículos Elétricos**: O script definir_eletricos.py é então utilizado para processar o rotas.rou.xml, gerando um ou mais arquivos de rotas modificados (ex: rotas_10_0_mod.rou.xml) onde uma porcentagem dos veículos é definida como elétrica.

8 **Primeira Execução (Coleta de Dados)**: O controlador_pa_opt.py é executado no modo first_run, usando um dos arquivos de rotas modificados. O objetivo desta fase é simular o tráfego sem postos de recarga para identificar as rotas mais populares entre os veículos elétricos. Os dados de visitação das lanes são salvos em um arquivo lane_visits.pkl.

9 **Alocação de Postos de Recarga**: Com base nos dados de visitação, o controlador_pa_opt.py aplica a heurística de alocação escolhida (Random, Greedy ou GRASP) para selecionar as melhores localizações para os postos de recarga. As localizações são salvas em um arquivo parking_areas.add.xml.

10 **Segunda Execução (Avaliação)**: O controlador_pa_opt.py é executado novamente, desta vez no modo second_run. Ele usa o mesmo arquivo de rotas modificado e o parking_areas.add.xml recém-criado. Esta simulação avalia a eficácia da alocação dos postos de recarga, medindo métricas como tempo de espera e distância até a recarga.

11 **Análise de Resultados**: Os resultados da segunda execução são anexados a um arquivo resultados_execucoes.csv. Este arquivo acumula os resultados de múltiplas execuções com diferentes parâmetros (heurísticas, número de PAs, porcentagem de VEs, etc.),

permitindo uma análise comparativa abrangente do desempenho das diferentes estratégias de alocação.

5. Conclusão

Este conjunto de scripts fornece uma estrutura flexível para pesquisar e otimizar a localização de infraestrutura de recarga para veículos elétricos em simulações de tráfego. A separação de funcionalidades entre os scripts, o uso de heurísticas variadas e a coleta detalhada de métricas permitem uma análise aprofundada e a tomada de decisões informadas sobre o planejamento de redes de recarga em ambientes urbanos.