/ attention /

{자지자지}

전비 모델 함수

```
# data Type = Mail / phone
int {Team HighFive_ProjectChaji .
return 0;}
//visit our web site
```

Main Algorithm

1. 온도에 따른 전비 변동 요인 계산

주행거리/전비가 온도에 따라 20~40%까지 차이 난다고 보고됨

겨울철 -10°C에서는 효율이 0.6~0.7까지 떨어지고, 여름철 35°C 이상에서 0.8 근처까지 하락.

<감쇠계수>

ALPHA_WINTER는 저온 환경에서 히터, 배터리 저온화, 난방 손실 등으로 인해 급격한 전비 저하를 반영.

ALPHA_SUMMER는 냉방 부담, 배터리 보호 시스템 작동 등으로 인한 완만한 전비 저하를 반영.

1. 온도에 따른 전비 변동 요인 계산

```
// 최적 온도 범위 설정 (18도~22도 사이가 전비 효율 최적 온도)
const OPTIMAL_TEMP_MIN = 18.0;
const OPTIMAL_TEMP_MAX = 22.0;

// 감쇠 계수 (겨울철이 여름보다 더 전비에 민감하다고 가정)
const ALPHA_WINTER = 0.015;
const ALPHA_SUMMER = 0.01;
```

```
export function getTemperatureWeight(temperature) {
  // 최적 온도 범위 내이면 가중치 1.0
  if (temperature >= OPTIMAL_TEMP_MIN && temperature <=
OPTIMAL_TEMP_MAX) {
    return 1.0;
  }

  // 최적 범위 밖일 경우, 얼마나 벗어났는지를 deviation으로 계산
  let deviation, alpha;
  if (temperature < OPTIMAL_TEMP_MIN) {
    deviation = OPTIMAL_TEMP_MIN - temperature; // 추운 쪽으로 벗어남
    alpha = ALPHA_WINTER; // 겨울 감쇠 계수 사용
  } else {
    deviation = temperature - OPTIMAL_TEMP_MAX; // 더운 쪽으로 벗어남
    alpha = ALPHA_SUMMER; // 여름 감쇠 계수 사용
  }

  // 가중치는 감쇠 함수: 1 / (1 + α * deviation^β)
  // 온도에서 벗어날수록 가중치가 감소하며, 곡선 형태로 줄어든다.
  return 1.0 / (1 + alpha * Math.pow(deviation, BETA));
}
```

Main Algorithm

2. 주행 방식에 따른 전비 변동 요인 계산

estimateCitySpeed()

→ 전체 시간 중 도심 구간의 속도를 추정.

cityBoost()

→ 도심 속도가 느릴수록 회생제동 효과로 전비가 좋아짐

calculateRoadWeight()

→ 전비 비율 * 도로 비율을 통해 전체 경로에서의 효율을 추정.

calculateRoadWeightByVehicle()

→ 차량의 도심/고속 전비 차이를 고려하여, 가중치를 차량별로 정규화해서 반영.

2. 주행 방식에 따른 전비 변동 요인 계산

```
export function estimateCitySpeed(
  cityDistance,
  totalDistance,
  totalTimeSeconds
) {
  if (totalDistance === 0 || totalTimeSeconds === 0) return 0.0; // 예외 처리

  // 전체 시간에서 도심 구간이 차지하는 비율만큼 도심 주행 시간 추정
  const cityTime = (cityDistance / totalDistance) * totalTimeSeconds;

  // 도심 평균 속도 = (도심 거리 km) / (도심 시간 시간 단위)
  return cityDistance / 1000 / (cityTime / 3600); // km/h
}
```

2. 주행 방식에 따른 전비 변동 요인 계산

```
export function calculateRoadWeightByVehicle(
  cityEv,
  highwayEv,
  cityDistance,
  highwayDistance,
  totalTimeSeconds
) {
  const totalDistance = cityDistance + highwayDistance;
  if (totalDistance === 0) return 0.0;

  // 도심과 고속 전비 평균값
  const averageEv = (cityEv + highwayEv) / 2;

  // 평균 전비 기준으로 상대 전비 가중치 계산 (비율화)
  const cityEvRatio = cityEv / averageEv;
  const highwayEvRatio = highwayEv / averageEv;

  // 도심 속도 추정 및 회생 제동 보정
  const cityTime = (cityDistance / totalDistance) * totalTimeSeconds;
  const citySpeed = cityDistance / 1000 / (cityTime / 3600); // km/h
  const adjustedCityEvRatio = citySpeed < 30 ? cityEvRatio * 1.35 :
cityEvRatio;
```

2. 주행 방식에 따른 전비 변동 요인 계산

```
  // 도심/고속 거리 비율
  const cityPortion = cityDistance / totalDistance;
  const highwayPortion = highwayDistance / totalDistance;

  // 거리 비율과 보정된 가중치의 가중 평균
  const roadWeight =
    adjustedCityEvRatio * cityPortion + highwayEvRatio * highwayPortion;

  return roadWeight;
}
```

```
*/ attention /*
```

{차지차지}

충전소 추천 알고리즘

```
# data Type = Mail / phone
int {Team HighFive_ProjectChaji .
return 0;}
//visit our web site
```

Haversine 공식

두 지점의 위도(latitude)와 경도(longitude)를 이용해, 지구 곡률을 고려한 구면 거리(직선 거리)를 계산하는 수학 공식

```
export default function haversineDistance(lat1, lon1, lat2, lon2) {
  const R = 6371000;
  const toRad = (deg) => (deg * Math.PI) / 180;
  const dLat = toRad(lat2 - lat1);
  const dLon = toRad(lon2 - lon1);
  const a =
    Math.sin(dLat / 2) ** 2 +
    Math.cos(toRad(lat1)) * Math.cos(toRad(lat2)) * Math.sin(dLon / 2) ** 2;
  const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
  return R * c;
}
screenLeft;
```

Main Algorithm

1. DB에서 모든 충전소를 메모리에서 로드

2. 필터링: 조건에 맞는 충전기만 추출

3. 충전소(statId)별로 그룹핑

4. 웨이포인트마다 반경 내 대표 충전소 추출 (속도 기준으로 정렬)

5. 웨이포인트 구간을 5개 영역(zone)으로 나누어 충전소 분배

6. 각 zone별 충전소 중 대표 충전소 선정 + 점수 높은 충전소 2개 추출 + statId 중복시 다시 뽑기

7. 전체 추천 충전소 반환

1. DB에서 모든 충전소를 메모리에서 로드

```
// [ 웨이포인트 기반 충전소 필터링 핵심 로직]
    private List<StationDTO> filterStations(List<LatLngDTO> waypoints,
double radiusMeters, boolean highwayOnly, StationFilterDTO filter) {
        // 1. DB에서 모든 충전소를 메모리로 로드
        Map<String, StationDTO> allChargers =
stationMemoryFromDBCache.getAll();
```

2. 필터링: 조건에 맞는 충전기만 추출

```
        // 3. 필터링: 조건에 맞는 충전기만 추출
        List<StationDTO> filteredChargers =
allChargers.values().stream()
                .filter(c -> shouldIncludeStation(c, highwayOnly)) // 고
속도로이면 급속만 허용
                .filter(c -> !"Y".equalsIgnoreCase(c.getDelYn())) // 삭
제된 충전기 제외
                .filter(c -> !freeParking ||
"Y".equalsIgnoreCase(c.getParkingFree()))
                .filter(c -> !noLimit ||
(!"Y".equalsIgnoreCase(c.getLimitYn()) && (c.getNote() == null ||
!c.getNote().contains("이용 불가"))))
                .filter(c -> {
                    try {
                        double o = Double.parseDouble(c.getOutput());
                        return o >= outputMin && o <= outputMax;
                    } catch (Exception e) {
                        return false;
                    }
                })
                .filter(c -> typeList.isEmpty() ||
typeList.contains(String.valueOf(c.getChgerType()).trim()))
                .filter(c -> providerList.isEmpty() ||
providerList.contains(c.getBusiId()))
                .toList();
```

3. 충전소(statId)별로 그룹핑

```
        // 4. 충전소(statId)별로 그룹핑
        Map<String, List<StationDTO>> groupedByStation = new HashMap<>
();
        for (StationDTO charger : filteredChargers) {
            groupedByStation.computeIfAbsent(charger.getStatId(), k ->
new ArrayList<>()).add(charger);
        }
```

4. 웨이포인트마다 반경 내 대표 충전소 추출 (속도 기준으로 정렬)

```
        // 5. 웨이포인트마다 반경 내 대표 충전소 추출 (속도 기준으로 정렬)
        Map<LatLngDTO, List<StationDTO>> wpToTopStations = new
LinkedHashMap<>();
        for (LatLngDTO wp : waypoints) {
            List<StationDTO> nearbyReps = new ArrayList<>();

            for (List<StationDTO> chargers : groupedByStation.values())
{
                chargers.sort(Comparator.comparingDouble(c -> -
Double.parseDouble(c.getOutput())));
                StationDTO rep = chargers.get(0); // 대표 충전기
                double dist = geoUtil.calcDistance(wp.getLat(),
wp.getLng(), rep.getLat(), rep.getLng());
                if (dist <= radiusMeters) {
                    rep.setDistance(dist); // 거리 저장
                    nearbyReps.add(rep);
                }
            }
            // 웨이포인트별 대표 충전소 중 상위 5개 추출 (사용자 선호 기반 점
수순)
            List<StationDTO> top5 = nearbyReps.stream()
                    .sorted((a, b) -> Integer.compare(
                            calculateWeightedSingleScore(b, priority),
                            calculateWeightedSingleScore(a, priority)))
                    .limit(5)
                    .toList();

            wpToTopStations.put(wp, top5);
        }
```

[사용자 선호 기반 평가 지표, 가중치 유틸]

```
private int calculateWeightedStationScore(List<StationDTO> chargers,
String priority) {
        if (chargers == null || chargers.isEmpty()) return 0;
        StationDTO rep = chargers.get(0);
        int speedScore = 0, reliabilityScore = 0, comfortScore = 0;

        try {
            double output = Double.parseDouble(rep.getOutput());
            speedScore = output >= 200 ? 6 : output >= 100 ? 5 : output
>= 50 ? 4 : output >= 7 ? 2 : 1;
        } catch (Exception ignored) {
        }
```

```
        int chargerCount = chargers.size();
        int maxChargerThreshold = 15;
        reliabilityScore += Math.min((chargerCount * 5) /
maxChargerThreshold, 5);
        if (rep.getStatUpdDt() != null &&
isWithinLast24Hours(rep.getStatUpdDt())) reliabilityScore += 2;

        if ("Y".equalsIgnoreCase(rep.getParkingFree())) comfortScore +=
2;
        if ("Y".equalsIgnoreCase(rep.getTrafficYn())) comfortScore += 2;
        if (rep.getUseTime() != null && rep.getUseTime().contains("24시
간")) comfortScore += 3;
```

```
        return switch (priority) {
            case "speed" -> speedScore * 2 + reliabilityScore +
comfortScore;
            case "reliability" -> speedScore + reliabilityScore * 2 +
comfortScore;
            case "comfort" -> speedScore + reliabilityScore +
comfortScore * 2;
            default -> speedScore + reliabilityScore + comfortScore;
        };
    }
```

5. 웨이포인트 구간을 5개 영역(zone)으로 나누어 충전소 분배

```
        // 6. 웨이포인트 구간을 5개 영역(zone)으로 나누어 충전소 분배
        int totalPoints = waypoints.size();
        int segment = Math.max(1, totalPoints / 5);
        Set<String>[] zones = new Set[]{new HashSet<>(), new HashSet<>
(), new HashSet<>(), new HashSet<>(), new HashSet<>()};

        for (int i = 0; i < totalPoints; i++) {
            LatLngDTO wp = waypoints.get(i);
            List<StationDTO> stations = wpToTopStations.getOrDefault(wp,
new ArrayList<>());
            int zoneIndex = Math.min(i / segment, 4);
            for (StationDTO s : stations)
zones[zoneIndex].add(s.getStatId());
        }
```

6. 각 zone별 충전소 중 대표 충전소 선정 + 점수 높은 충전소 2개 추출 + statId 중복시 다시 뽑기

```
        // 7. 각 zone별 충전소 중 대표 충전소 선정 + 점수 높은 충전소 2개 추
출 + statId 중복시 다시 뽑기
        List<StationDTO> result = new ArrayList<>();
        Set<String> addedStatIds = new HashSet<>();

        for (Set<String> zone : zones) {
            List<StationDTO> zoneList = zone.stream()
                    .map(groupedByStation::get)
                    .filter(Objects::nonNull)
                    .map(list -> list.stream()
                            .max(Comparator.comparingDouble(c ->
safeParseOutput(c.getOutput())))
                            .orElse(null))
                    .filter(Objects::nonNull)
                    .toList();

            // 점수순 정렬
            List<StationDTO> topByScore =
selectTopStationsByScore(zoneList, groupedByStation, zoneList.size(),
priority);

            int added = 0;
            for (StationDTO s : topByScore) {
                if (added >= 2) break;
                if (!addedStatIds.contains(s.getStatId())) {
                    result.add(s);
                    addedStatIds.add(s.getStatId());
                    added++;
                }
            }
        }
```