CJ 대한통운 미래기술 챌린지 2023

실시간 주문 대응 Routing 최적화
알고리즘 설명 자료

목차

1. 아이디어 3
2. 구동환경 3
3. 데이터 전처리 4
   3.1. 주문 데이터 전처리 4
   3.2. 터미널 좌표 및 각도 정보 계산 4
4. 객체 설명 4

4.1. Order
4

4.2. Vehicle
5

4.3. Terminal
6

5. 알고리즘 개요
6

5.1. 배송 우선순위
6

5.2. 하차 후 차량 복귀 알고리즘
8

5.3. 시뮬레이션
8

6. 알고리즘 설명
12

6.1. 내부 함수
12

6.2. 배송 처리 함수
14

1. 아이디어

- Routing 전략

차량의 적재 최대 가능 용량(maxCBM)과 주문의 적재용량(CBM)을 고려하였을 때, 차량에 실을 수 있는 주문의 수는 제한적이고 이에 Routing의 경로가 비교적 단순합니다. 따라서 경로 최적화보다는 동일 목적지 배송의 경우 적재 시간이 (동일 목적지 배송건 수 - 1) * 60분만큼 절감된다는 제약 조건에 주안점을 두어 큰 차량을 우선 활용한 동일 목적지 배송 건들의 공동 배송 최적화에 비중을 두었습니다. 주문 데이터들을 총 4가지 유형 (1. 동일 터미널(상차지) - 동일 목적지 / 2. 다른 터미널 - 동일 목적지 / 3. 동일 터미널 - 다른 목적지 / 4. 다른 터미널)으로 분류하여 각 유형을 타겟팅한 알고리즘을 개발하였고 표시된 우선 순위 별로 주문을 처리하였습니다.

- 차량 복귀 전략

모든 차량이 하차 작업 완료 후 상차 터미널과 다른 터미널로 복귀가 가능합니다. 터미널에서 목적지 간의 거리가 상당한 주문이 존재하기 때문에 동일 터미널로의 복귀는 이동 비용의 손해를 감수해야 합니다. 따라서 차량이 유동적으로 다른 터미널로 복귀 가능토록 설계하였고, 차량의 쏠림 현상을 방지하고 수요에 따라 적절히 분배하기 위해 설정한 여러 지표들 가운데 각 터미널 까지의 거리 [=비용] x (해당 터미널의 예상 복귀 차량 + 현재 상주 차량)[=가용 가능 차량] / 해당 터미널의 주문 수[=수요]라는 비용, 차량 수, 수요를 고려한 지표 기준으로 복귀 터미널을 선택하여 차량을 복귀시켰습니다.

- 72시간 내 모든 주문 처리 전략

시간의 흐름에 따른 주문 패턴의 변화로 차량 쏠림 및 부족 현상을 맞닥뜨렸고 이로 인해 데드 라인 주문들의 미처리 가능성이 존재하였습니다. 따라서, 특정 시간 간격(interval)을 강제로 설정하여 활성화된 차량이 가장 많은 곳(할당된 주문이 많지 않아 차량의 잉여가 발생하는 곳)에서 차량 부족이 극심히 예상되는 곳으로 차량을 이동시켜 미처리 주문이 발생하지 않도록 방지하였습니다.

2. 구동환경

Google Colab Pro 환경에서 진행하였으며 사용한 모듈은 다음과 같습니다.

- pandas: 데이터 처리 및 분석을 위한 라이브러리
- numpy: 수치 계산을 위한 라이브러리
- datetime: 날짜와 시간 처리를 위한 모듈
- queue.PriorityQueue: 우선순위 큐를 위한 클래스
- math: 수학적 연산을 위한 모듈
- itertools.combinations: 조합 생성을 위한 함수
- itertools.permutations: 순열 생성을 위한 함수

- collections.defaultdict: 딕셔너리의 기본값을 설정하는 클래스
- google.colab.drive: Google Colab에서 드라이브를 마운트하는 라이브러리

또한 공유 드라이브를 사용해 함께 작업을 진행했으며 데이터 파일 경로는 “/content/drive/Shareddrives/CJ 대한통운 공모전/용탁/data/”에 위치합니다.

3. 데이터 전처리

3.1. 주문 데이터 전처리

- 주문 데이터에서 필요한 컬럼을 선택하고 이름을 재지정합니다.
- 주문 시간을 시간 단위로 변환하여 새로운 컬럼에 저장합니다.
- 출발 터미널과 목적지 터미널 간의 거리와 시간 정보를 가진 데이터와 주문 데이터를 조인합니다.
- 날짜와 그룹을 기반으로 0~23까지의 그룹 번호를 할당하는 함수를 정의하여 적용합니다.

3.2. 터미널 좌표 및 각도 정보 계산

- 터미널 데이터에서 필요한 컬럼을 선택하여 딕셔너리로 저장합니다.
- 주문 데이터의 목적지 터미널에 대한 좌표 정보를 딕셔너리로 저장합니다.
- 주문 데이터와 터미널 데이터를 활용하여 각 주문의 하역 타임 윈도우 [(하역가능시작, 하역가능마감) 형태로 표현한 리스트] 및 터미널을 원점으로 북쪽 기준 목적지까지 시계 방향 각도를 계산합니다.

4. 객체 설명

결과 테이블 작성을 위해 주문 정보, 차량 정보, 터미널 정보를 담은 Order, Vehicle, Terminal 객체를 생성하였습니다.

4.1. Order

주문 정보를 나타내는 클래스로, 다양한 속성과 동작을 포함합니다. 주요 속성으로는 주문 번호(ID), 적재 용량, 출발 터미널 ID, 목적지 ID, 하차 가능 시간대 등이 있습니다. 주문 객체는 다양한 배송 전략에 따라 업데이트되며, 주문 처리 상태, 도착 및 출발 시간 등을 관리합니다.

- updateForSTSD(vehId, seq)
  주문을 담당할 차량 ID와 시퀀스를 받아와 업데이트합니다. 동일 터미널, 동일 목적지 주문을 처리하는 경우 동작합니다.

- updateForSTDD(vehId, seq, plusTime)

차량 ID, 시퀀스, 추가 시간을 받아와 업데이트합니다. 같은 목적지가 아닌 주문을 처리하는 경우 동작합니다.

- updateForDTSD(vehId, seq, plusTime)
  차량 ID, 시퀀스, 추가 시간을 받아와 업데이트합니다. 다른 터미널로 이동하는 경우 동작합니다.

- updateForDTDD(vehId, seq, plusTime)
  차량 ID, 시퀀스, 추가 시간을 받아와 업데이트합니다. 다른 터미널, 다른 목적지 경로로 이동하는 경우 동작합니다.

- updateForUrgent(vehId, seq):
  차량 ID, 시퀀스를 받아와 업데이트합니다. 긴급 배송 시 동작합니다.

- __str__()
  주문 객체의 속성을 디버깅 및 정보 확인용으로 출력합니다.

4.2. Vehicle

차량 정보를 나타내는 클래스로, 차량의 톤급, 최대 용량, 고정 및 가변 운행비 등을 포함합니다. 차량은 주문을 처리하고 라우팅 결과를 관리하며, 주행 거리, 총 이동 및 하역 시간 등을 추적합니다.

- updateForSTSD(samedesID)
  동일 터미널, 동일 목적지 주문들을 받아와 업데이트합니다. 주문을 묶어서 처리하는 경우 동작합니다.

- updateForSTDD(path)
  동일 터미널, 다른 목적지 주문 경로를 받아와 업데이트합니다. 주문마다 상차와 하차를 다르게 처리하는 경우 동작합니다.

- updateForDTSD(path)
  다른 터미널, 같은 목적지 주문 경로를 받아와 업데이트합니다. 다른 터미널로 이동하는 경우 동작합니다.

- updateForDTDD(path)
  다른 터미널, 다른 목적지 주문 경로를 받아와 차량이 활성화되어 있다면 할당해 주고, 차량이 없다면 새로운 차량을 구매하여 동작합니다.

- updateForUrgent(order)
  기간이 촉박한 긴급 유보 배송건을 위한 함수로 입력받은 order를 처리하기 위해 차량을 출차시키고, 차량이 없다면 구매하여 동작합니다. 동시에 결과 테이

블을 위해 dataframe에 row를 추가하며 차량의 작업 완료 시점을 update합니다.

- updateForStarvation(top1Ter)
  차량이 가장 필요한 터미널을 받아와 해당 터미널로 차량을 보내줄 때 동작하며 차량 정보를 해당 터미널로 이동함에 맞추어 업데이트합니다.

- __str__()
  차량 객체의 속성을 디버깅 및 정보 확인용으로 출력합니다.

4.3. Terminal

터미널 정보를 나타내는 클래스로, 주문 처리 및 차량 상태를 관리합니다. 터미널에는 주문 딕셔너리, 활성 및 비활성 차량 목록, 현재 차량 대수 등이 포함됩니다.

- __init__(id, latitude, longitude)
  터미널을 초기화하고 속성을 설정합니다.

- 기타 관리 메서드
  주문 및 차량 목록 관리에 사용되는 메서드들이 있습니다.

5. 알고리즘 개요

5.1. 배송 우선순위

현재 많이 사용하는 Hub-and-Spoke 방식으로 구현하고자 생각했습니다. 하지만 해당 방식 대신 터미널에서 직배송 상황만을 가정하는 문제이기 때문에 다음과 같은 우선순위를 두고 차례로 묶음 배송을 해 효율적으로 처리하고자 했습니다.

5.1.1. 출발 터미널과 목적지가 모두 같은 주문 (SameTerSameDest)

이 상황에서는 출발 터미널과 목적지가 모두 동일한 주문들을 공동배송으로 묶어 효율적으로 처리합니다.

주문을 가장 큰 크기부터 시작하여, 같은 트럭에 담을 수 있는 주문을 순차적으로 확인합니다. 이때, 해당 주문들의 총 용량이 트럭의 최대 용량을 넘지 않도록, 주문 총 합의 CBM이 20을 넘도록 합니다. 만약 한 대의 트럭으로 담을 수 없는 주문이 있다면, 그 주문은 따로 보류하고, 다음으로 큰 크기의 주문을 추가하여 공동 배송을 시도합니다.

5.1.2. 출발 터미널이 다르지만 목적지가 같은 배송 (DiffTerSameDest)

이 경우에는 서로 다른 터미널 출발이지만, 같은 목적지로 가는 주문들을 효율적인 경로로 배송할 수 있도록 합니다.

주어진 주문들 중에서 가능한 모든 조합을 계산하여, 주문들을 효율적으로 그룹화합니다. 이때, 주문의 총 용량이 트럭의 최대 용량을 넘지 않는 조합을 찾습니다. 계산된 각 그룹에 대해 최적 경로를 계산하고, 효율적인 경로를 선택합니다. 이때, 주문들 간의 거리와 시간 정보를 활용하여 최적 경로를 결정합니다.

5.1.3. 출발 터미널이 같지만 목적지가 다른 주문 (SameTerDiffDest)

이 상황에서는 출발 터미널은 동일하지만, 주문들의 목적지가 서로 다른 경우에 대한 알고리즘을 살펴보겠습니다. 이 알고리즘은 주어진 주문들을 효율적으로 처리하고, 차량을 출발 터미널로 복귀시키는 전략을 구현합니다.

이 상황에 해당하는 주문들 중 가능한 모든 조합을 계산하여, 가장 효율적인 경로를 찾아 묶음 배송을 하도록 합니다. 그 중 최대 용량을 넘지 않으며 묶음 배송을 했을 때 배송 시간 안에 배송가능한 조합을 찾아 최적 경로를 결정합니다.

5.1.4. 출발 터미널이 다르지만 목적지가 다른 주문 (DiffTerDiffDest)

이 경우에는 터미널이 다르지만 목적지가 가까운 경우나 경로가 유사한 경우에 대한 전략을 구현합니다.

이때, 배송 경로의 위도와 경도를 기반으로하여 각도를 측정합니다. 출발 터미널과 목적지의 중심 위도 및 경도 값이 비슷한 주문들을 묶어서 처리합니다. 이를 위해 주문 객체를 처리되지 않은 전체 주문 객체 중에서 적재 용량(CBM)이 55를 넘지 않는 두 개의 주문 객체를 조합합니다. 또한, 주문들의 경로 각도 차이가 일정한 threshold 안에 들어가도록 주문 객체 조합을 선별합니다. 선별된 조합 중 배송 가능 시간 안에 하역할 수 있는 조합을 찾아 경로를 결정합니다.

5.1.5. 최종 유보 처리 알고리즘

이 알고리즘은 여러 주문들 중 위의 4가지에 해당하지 않아 유보된 모든 주문을 처리하기 위한 긴급 배송 알고리즘입니다. 마감 시간이 가장 빠른 주문의 처리를 우선적으로 보장하는 전략을 구현합니다. 마감 시간이 가장 빠른 주문 건은 단독으로 배송 처리됩니다. 이때, 해당 주문의 용량(CBM)을 최대한 맞추어서 차량을 선택하여 배송합니다.

5.2. 하차 후 차량 복귀 알고리즘

하차 후 차량이 터미널로 이동하는 알고리즘은 효율적인 터미널 선택을 통해 차량의 이동 거리와 시간을 최소화하는 전략을 적용합니다. 주문이 몰리는 시간 대 이전까지는 차량이 가장 부족한 터미널을 거리를 고려하여 선택하고 해당 터미널로 차량을 복귀시킵니다. 주문이 많이 몰리는 시간 대 이후에는 우선 차량을 부족한 터미널에 보충하는 것이 시급하므로 거리를 고려하지 않고 최우선순위 터미널로 보냅니다.

5.3. 시뮬레이션

5.3.1. Event type - 사건 종류

1. `orderByTime` (주문 시점에 도달 시)
   이 단계는 주문이 생성되어 각 터미널에 할당되는 시점을 나타냅니다. 주문 데이터의 시간 정보를 활용하여 각 주문의 주문 시간(group)이 설정되고 터미널별로 보유 차량을 통해 Routing을 실행합니다.

2. `departDestinaion` (목적지에서 최종 하차 작업 완료 시)
   해당 단계는 차량이 최종 목적지에 도착하여 주문의 하차 작업을 완료하는 시점을 의미합니다. 현재 시각, 주문의 상차지/목적지 방문 순서와 작업 소요 시간을 계산하여 최종 목적지에서 하차 작업을 완료한 시간이 설정됩니다.

3. `imBack` (운행 차량의 터미널 복귀 시)
   이 시점은 운행 중인 차량이 최종 목적지에서 모든 주문 작업을 완료하고 복귀 룰에 의해 지정된 터미널로 복귀하는 시점을 나타냅니다. 복귀 시점은 하차 작업 완료 시간에서 해당 터미널로 돌아오는 시간을 계산하여 설정됩니다.

4. `deadline` (가장 처리가 시급한 주문의 데드라인 시)
   이 단계는 시뮬레이션에서 가장 긴급하게 처리해야 할 주문의 데드라인 시간(72시간 내 처리를 위해 출발해야 하는 터미널 출발 시간 데드라인)을 의미합니다. 유보된 주문들의 데드라인 시간 중 가장 빠른 시간으로 설정됩니다.

5. `recoverStarvation` (정해진 interval 시)
   이 단계는 정해진 interval마다 상주 활성화 차량이 가장 많은 터미널에서 수요가 가장 급한 터미널(현재 터미널의 상주 활성화 차량 + 터미널로 도착 예정인 차량 / 현재 해당 터미널의 처리해야 할 전체 주문 수, 작은 순) 또는 차량 대수가 가장 적은 터미널 (현재 터미널의 상주 활성화 차량 + 터미널로 도착 예정인 차량 + 구입 가능한 차량)로 차량을 강제 분배 시킵니다.

6. `end` (시뮬레이션 종료)
   이 시점은 시뮬레이션의 종료를 나타냅니다. 시뮬레이션 동작이 끝나는 지점을 설정하여 시뮬레이션이 완료되는 시점을 나타냅니다.

이러한 다양한 시점과 단계는 시뮬레이션 프로세스 내에서의 중요한 시간 정보를 나타내며, 각각의 단계는 해당 시뮬레이션의 흐름과 동작을 제어하는 데 사용됩니다.

5.3.2. Architecture - 시뮬레이션 구조

이 구조는 시뮬레이션 프로그램의 주요 구성 요소와 실행 흐름을 설명합니다.

1. `main` 흐름
   `initialize()` 함수를 호출하여 시뮬레이션 시간을 나타내는 변수(시계) `newClock`과 각 사건 별로 가장 빨리 실행해야 할 사건의 시간을 담은 `timestamp`를 생성합니다. 시스템을 초기화한 후, `timing()` 함수를 호출하여 현재 시뮬레이션 시간에서 가장 가까운 미래의 사건을 찾아 해당 사건의 `event_type`을 반환하고 관련된 함수를 실행한 뒤 timestamp와 객체/변수 상태를 업데이트하는 과정을 end()에 도달할 때까지 반복합니다.

2. `initialize()` 함수
   시뮬레이션 시작 전에 시스템을 초기화하는 함수입니다. 이 함수에서는 초기 상태 설정, 변수 초기화, 필요한 데이터 구조 생성 등이 수행됩니다. `newClock`과 `timestamp`를 생성한 뒤, group 0에 해당하는 주문을 생성하여 Routing한 뒤, 해당 사건과 관련된 것들을 업데이트합니다.

3. `timing()` 함수
   가장 가까운 미래의 이벤트를 찾아 그에 대한 정보를 반환하는 함수입니다. 이 함수는 newClock를 업데이트하고, timestamp에서 가장 빠른 시간을 찾아 다음으로 발생할 이벤트가 어떤 종류인지 결정하여 반환합니다.

4. `if event_type == i:`
   시뮬레이션에서 발생한 이벤트 타입에 따라 해당 이벤트와 관련된 함수를 실행하는 조건문입니다. 각각의 이벤트 타입에 따라 해당 이벤트의 처리를 위한 함수가 호출됩니다.

위와 같은 구조를 통해 시뮬레이션은 초기화 단계를 거쳐 시간을 추적하며 다양한 이벤트가 발생할 때마다 해당 이벤트에 대응하는 함수를 실행하여 시뮬레이션의 동작을 시뮬레이션 시간에 따라 제어합니다.

5.3.3. 함수 설명

- initialize()

시뮬레이션 시작 시 시스템의 초기 상태를 설정하고, 터미널과 주문 객체의 초기 분류를 수행하여 시뮬레이션의 동작을 준비합니다.

1. `nowClock` 초기화
   시뮬레이션 시계인 `nowClock`을 생성하여 0으로 세팅합니다.

2. 차량 리스트 정렬
   비활성 상태의 차량 리스트(`terminal.inactivevehicleList`)를 차량의 CBM 크기 순서대로 정렬합니다. 이를 통해 크기 순서대로 차량을 호출할 시 0번째 원소부터 추출하여 정렬을 반복하는 것을 피합니다.

3. 주문 객체 분류
   주문 객체들은 그룹별로 주문 시간(orderbytime의 group)으로 분류되어 있습니다. 이 중 그룹 0인 주문 객체들을 터미널 객체의 `orderDict` 안에 목적지 별로 분류하여 저장합니다. 각 터미널마다 자신의 `orderDict`에 해당 정보를 저장하여 관리하게 됩니다.

4. `totalRoutingAlgorithm()` 호출 및 관련 state/timestamp 업데이트
   해당 함수를 호출하여 차량 객체에 주문을 할당하고 관련된 객체들의 상태와 영향을 주는 사건들의 시간을 업데이트합니다. 주문 할당 시 경로가 확정되기 때문에 작업완료와 관련된 `departDestination` 함수의 변수(UnloadedComplete,PriorityQueue), 그리고 타임스탬프(timestamp[1] = UnloadedComplete의 가장 선두 객체)를 업데이트하여 시뮬레이션 동작을 조정하는 역할을 합니다.

- Orderbytime()

시뮬레이션 내에서 시간대에 따른 주문 처리와 관련된 동작을 수행합니다. 또한 각 주문을 터미널의 orderDict에 목적지 별로 분류하여 저장하고 시뮬레이션 동작을 관리합니다.

1. 주문 객체 분류
   UpdateTerByOrderDesID 함수를 실행하여 현재 시뮬레이션 시계 nowClock에 해당하는 그룹에 속하는 주문 객체들을 각 터미널 객체의 orderDict 안에 목적지 별로 분류하여 넣어줍니다.

2. `totalRoutingAlgorithm()` 호출 및 관련 state/timestamp 업데이트
   해당 함수를 호출하여 차량 객체에 주문을 할당하고 관련된 객체들의 상태와 영향을 주는 사건들의 시간을 업데이트합니다. 주문 할당 시 경로가 확정되기

때문에 작업완료와 관련된 `departDestination` 함수의 변수 (UnloadedComplete,PriorityQueue), 그리고 타임스탬프(timestamp[1] = UnloadedComplete의 가장 선두 객체)를 업데이트하여 시뮬레이션 동작을 조정하는 역할을 합니다.

3. ordertable 출력
   주문 테이블을 출력하는 단계로, 시뮬레이션 진행 상황을 확인할 수 있습니다.

- departDestination()

시뮬레이션 내에서 가장 빠른 작업 완료 시간을 가진 차량의 복귀 알고리즘을 실행하고, 차량을 복귀시키며 관련한 타임스탬프를 업데이트합니다.

1. timestamp[1]에 해당하는 차량 선택
   timestamp[1]에서 가장 우선순위가 높은(가장 빠른 작업 완료 시간을 가진) 차량을 가져옵니다.

2. 차량의 터미널 복귀 알고리즘 실행
   2-1) ~6일차 : focus on 각 터미널의 수요 대응성 및 비용을 고려한 복귀
   전체 터미널 목록 중에서 각 터미널 까지의 거리 * (해당 터미널의 예상 복귀 차량 + 현재 상주 차량) /해당 터미널의 주문 수를 기준으로 복귀 터미널을 선택하고 차량을 복귀시킵니다.

   2-2) 7일차 : focus on Deadline 주문 처리
   각 터미널의 데드라인에 가까운 주문들을 안정적으로 처리하기 위하여 전체 터미널에서 각 터미널의 예상 복귀 차량 + 현재 상주 차량 + 구입가능한(비활성화)차량이 가장 적은 터미널로 차량을 복귀시킵니다.

3. 차량 터미널 복귀 timestamp 업데이트
   선택한 차량의 차량 터미널 복귀 시간을 터미널 복귀와 관련된 `imBack` 함수의 변수(headToTer, PriorityQueue)에 업데이트하고, 가장 빠른 작업 완료시간 (timestamp[2] = headToTer의 가장 선두 객체)을 timestamp[2]에 업데이트합니다.

- imBack()

시뮬레이션 내에서 가장 빠른 터미널 복귀 시간을 가진 차량의 복귀 상태를 업데이트하고, 해당 터미널에 주문이 존재하고 해당 주문이 배송 가능할 시 이를 할당합니다.

4. timestamp[2]에 해당하는 차량 선택
   timestamp[2]에서 가장 우선순위가 높은(가장 빠른 복귀 터미널 도착 시간을 가진) 차량을 가져옵니다.

5. 차량 터미널 복귀 관련 state/timestamp 업데이트
   차량의 현재위치, 터미널의 현재 상주 차량 수 etc.

6. `generalRoutingAlgorithm()` 호출 및 관련 상태/timestamp 업데이트
   복귀 차량을 가지고 배송할 수 있는 가장 큰 사이즈의 주문을 처리합니다.

- processDeadline()

  모든 주문의 deadline이 할당되기 때문에 `processDeadline`함수의 변수 deadlineList(headToTer, PriorityQueue)의 선두 객체가 delivered 되었다면 다음 선두 객체의 시간으로 timestamp[3]으로 업데이트하고 not delivered라면 urgentDelivery(긴급배송 알고리즘)을 실행합니다.

- recoverStarvation()
  정해진 interval마다 상주 활성화 차량이 가장 많은 터미널에서 수요가 가장 급한 터미널(현재 터미널의 상주 활성화 차량 + 터미널로 도착 예정인 차량 / 현재 해당 터미널의 처리해야 할 전체 주문 수, 작은 순) 또는 차량 대수가 가장 적은 터미널 (현재 터미널의 상주 활성화 차량 + 터미널로 도착 예정인 차량 + 구입 가능한 차량)로 차량을 강제 분배 시키고 관련된 변수(headToTer)과 timestamp들을 업데이트합니다.
- end()
  결과 테이블을 출력합니다.

6. 알고리즘 설명

6.1. 내부 함수

▶ is_overlap (ranges, target_range)
: target_range와 ranges 사이에서 겹치는 time window가 있는지 확인하는 함수

▶ sum_variable_in_class_list (class_list, variable_name)
: class_list의 클래스 원소들의 variable_name 값을 전부 더해주는 함수

▶ is_within_ranges (value, tuple_list)

: time window 정보를 list로 담고 있는 tuple_list에 value가 있는지 확인하는 함수

▶ flatten_dict_values (dictionary)
: 주문을 할당받은 터미널 객체 내의 orderDict를 list로 바꿔주는 함수

▶ classifyOrder1 (order)
: sameTerSameDest에 해당하는 주문 객체 리스트를 (목적지, 터미널)을 key 값으로 갖는 딕셔너리로 저장해주는 함수

▶ classifyOrder2 (order)
: diffTerSameDest에 해당하는 주문 객체 리스트를 목적지를 key값으로 갖는 딕셔너리로 저장해주는 함수

▶ classifyOrder3 (order)
: sameTerDiffDest에 해당하는 주문 객체 리스트를 터미널을 key값으로 갖는 딕셔너리로 저장해주는 함수

▶ commonDistribute (data, curCBM)
: sameTerSameDest 알고리즘에 사용하는 내부 함수로 최대, 최저 CBM을 고려한 공동배송지 리스트를 return하는 함수

▶ PossibleOrder (timenow, orderlist)
: 주문 객체 리스트인 orderlist에서 현재 시점으로부터 해당 객체에 있을 때 time window를 고려하여 배송 가능한 주문들을 출력해주는 함수

▶ PossibleCombination (OrderList)
: diffTerSameDest, sameTerDiffDest에서 사용하는 내부함수로 주문 리스트에 대한 모든 경로를 조합하는 함수

▶ CapacitiyLimitCombination (OrderList, capacity = 55)
: 위의 코드에서 실행된 결과를 바탕으로 CBM을 고려한 조합 선별, capacity 값을 설정해 주지 않는다면 55가 기본값

▶ RoutePossible_Check3 (now, List)
: sameTerDiffDest에서 사용하는 함수로 현재 주어진 객체들의 조합 경로가 가능한지 True, False를 반환해주는 함수

▶ RoutePossible_Check2 (now, List)
: diffTerSameDest에서 사용하는 함수로 현재 주어진 객체들의 조합 경로가 가능한지 True, False를 반환해주는 함수

▶ PossibleRoute3 (OrderList, capacity=55)

: CapacitiyLimitCombination(), RoutePossible_Check3() 함수들을 이용하여 TimeWindow와 CBM을 모두 고려하여 sameTerDiffDest에서의 경로 산출

▶ PossibleRoute2 (OrderList, capacity=55)
: CapacitiyLimitCombination(), RoutePossible_Check2() 함수들을 이용하여 TimeWindow와 CBM을 모두 고려하여 diffTerSameDest에서의 경로 산출

▶ check_duplicates (list1, tuple2)
: list1과 tuple2의 모든 원소에 중복값이 있는지 확인해주는 내부함수, FindMostLeastCost() 함수에서 사용

▶ check_len (routelist)
: 리스트 내에 있는 튜플 원소의 수를 구하는 함수

▶ Calcul_Distance2 (List)
: diffTerSameDest에 대한 각 튜플에 대한 거리를 계산하는 함수, FindMostLeastCost() 함수에서 사용

▶ Calcul_Distance3 (List)
: sameTerDiffDest에 대한 각 튜플에 대한 거리를 계산하는 함수, FindMostLeastCost() 함수에서 사용

▶ FindMostLeastCost (List, route, n)
: 1차 조합된 경로들인 route와 diffTerSameDest, sameTerDiffDest를 각각 n = 2,3으로 입력받아 실행, 중복이 없는 튜플들을 찾아 총 거리 및 비용을 계산하고 가장 최적의 경로를 찾아 return

6.2. 배송 처리 함수

totalRoutingAlgorithm() 함수로 처리되며, 해당 함수는 다음과 같은 구조입니다.

```
def totalRoutingAlgorithm():
    sameTerSameDest()
    difTerSameDest()
    sameTerDifDest()
    difTerDifDest()
```

위 totalRoutingAlgorithm() 실행 전, imBack() 함수로 인해 모든 차량들이 CBM 순서로 정렬되어 있음은 보장됩니다.

1) sameTerSameDest():

- 실행 과정:
  모든 터미널 확인 -> CBM 큰 차량부터 배송 -> 배송 가능 시간 확인 -> 배송 및 객체 정보 업데이트

- 알고리즘 설명:
  터미널 객체는 주문 정보 딕셔너리를 추가로 가집니다. 이 주문 정보 딕셔너리는 목적지를 키로, 해당 목적지에 대한 주문 객체들을 배열로 가집니다. 해시 구현이므로 접근에는 약 O(1)이 소요되도록 하여 시간을 절약합니다. 이후, 활성화 차량부터 CBM이 제일 큰 차량을 선택하고, 해당 조건에 맞는 주문들을 정렬 후, 차량 CBM에 맞게 주문들을 선택합니다.

2) difTerSameDest():

- 실행 과정:
  모든 터미널 확인 -> 터미널 내 전체 주문 가져오기 -> 목적지 별 묶음 주문 분류 -> 배송 경로 모음들을 생성 -> 최적 경로 모음 선택 -> 배송 및 객체 정보 업데이트

- 알고리즘 설명:
  터미널 객체로부터 모든 주문 리스트를 가져옵니다. 각 주문들을 묶어서(2건 이상) 같은 목적지 별로 딕셔너리에 정리합니다. 묶인 주문들로 가능한 경로 모음을 생성합니다. (각 경로 모음 내 경로 1개마다 차량이 처리합니다.). 경로 모음 중 최적의 경로 모음을 선택하고 시작 터미널에서 차량을 보내며 처리합니다. 차량이 없으면 처리하지 않습니다. 경로 모음을 선택할 때, 조건(배송 가능 시간)에 따라 미리 필터링하여 경로 조합 경우의 수를 줄였습니다.

3) sameTerSameDest():

- 실행 과정:
  모든 터미널 확인 -> 터미널 내 전체 주문 가져오기 -> 목적지 별 묶음 주문 분류 -> 배송 경로 모음들을 생성 -> 최적 경로 모음 선택 -> 배송 및 객체 정보 업데이트

- 알고리즘 설명:
  터미널 객체로부터 모든 주문 리스트를 가져옵니다. 각 주문들을 묶어서(2건 이상) 같은 목적지 별로 딕셔너리에 정리합니다. 묶인 주문들로 가능한 경로 모음을 생성합니다. (각 경로 모음 내 경로 1개마다 차량이 처리합니

다.). 경로 모음 중 최적의 경로 모음을 선택하고 시작 터미널에서 차량을 보내며 처리합니다. 차량이 없으면 처리하지 않습니다. 경로 모음을 선택할 때, 조건(배송 가능 시간)에 따라 미리 필터링하여 경로 조합 경우의 수를 줄였습니다.

4) difTerDifDest():

- 실행 과정:

  주문 목적지의 각도 계산 -> 위 과정에서 처리되지 않은 모든 주문에 대하여 CBM을 고려한 객체 조합(2개씩) 선택 -> 선택된 객체 조합에서 다시 각도를 기준으로 조합 선별 -> 선택된 객체 조합에서 다시 거리 차를 고려하여 조합 선별 -> 선택된 조합들로 조건에 맞게 최적 경로를 생성 -> 배송 및 객체 정보 업데이트

- 알고리즘 설명:

  주문 객체에 북쪽을 기준으로 각 주문의 직배송 선분 각도를 기록합니다.
  이후 CBM 55를 기준으로 가능한 두가지 주문 조합들을 전부 생성합니다. 3가지 이상의 묶음 배송 조합을 선별하지 않는 이유는 실행 시간이 길어질 것을 염려했기 때문입니다. 이후 각도를 기준으로 재선별하고, 그것들에서 다시 각 주문 조합의 터미널 사이 거리, 목적지 사이 거리를 미리 설정한 한계값을 초과하지 않은 조합들만 추려냅니다. 이후 최종 선별된 주문 조합들에 대해 각 조합마다 경로를 생성해보며 최적을 찾고, 최적 경로들에 대해 차량이 전부 소진될 때까지 혹은 주문 조합이 소진될 때까지 그 최적 경로로 차량을 보내 배송합니다.
  이미 배송된 경로 조합은 배송하면 안되므로 visited 딕셔너리로 처리 여부를 확인했습니다.