OGC 2024 묶음배송 최적화 알고리즘

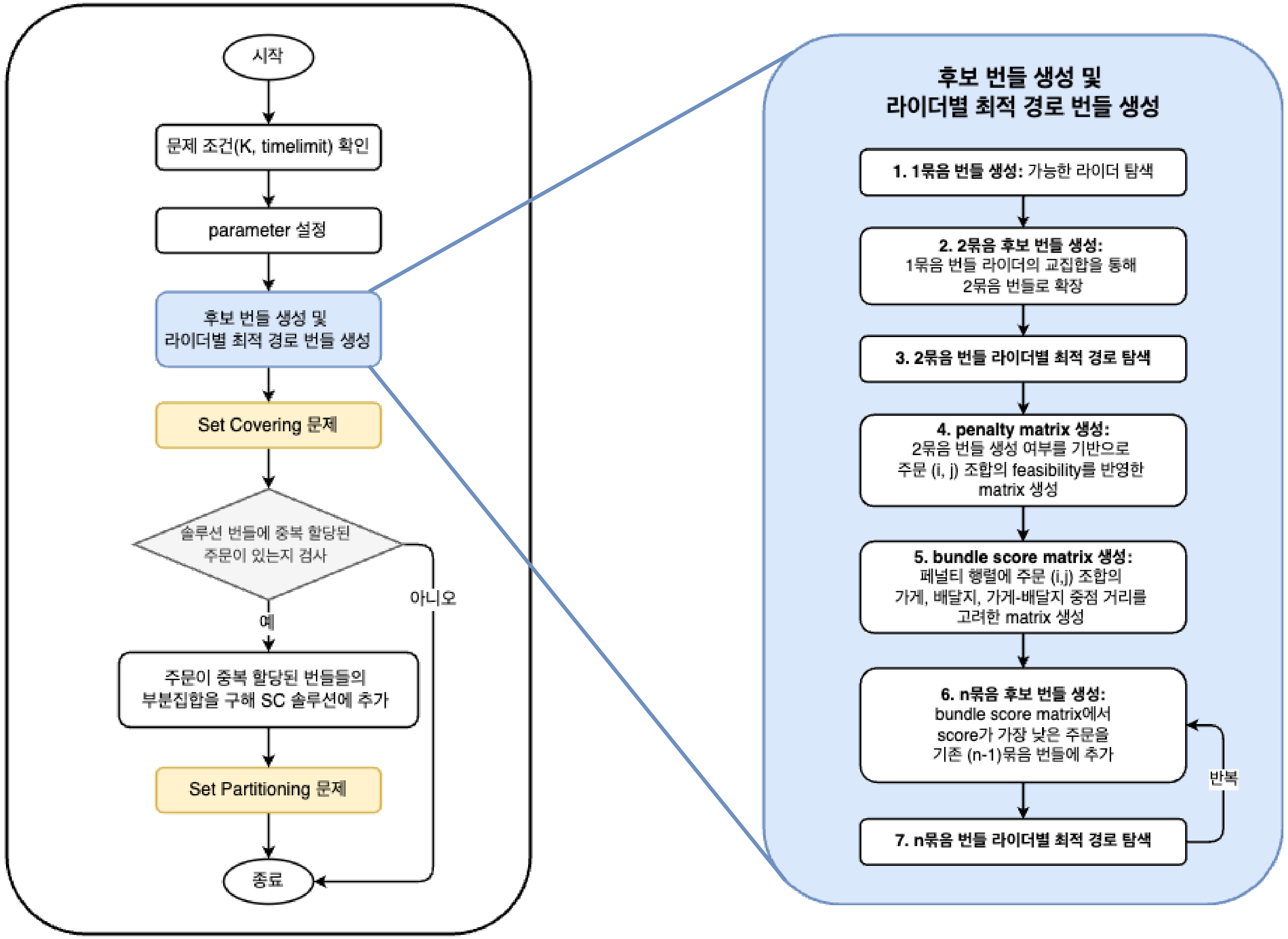
Slashe 최서여, 정보연

Dept. of Industrial and Management Engineering,
Hankuk University of Foreign Studies

Contents

1. 알고리즘 로직

알고리즘 전체 Flowchart



1. 알고리즘 로직

1. 번들 생성 > 2. 최적 경로 도출 > 3. Set Covering

후보 번들 생성 및 라이더별 최적 경로 번들 생성

1. **1묶음 번들 생성:** 가능한 라이더 탐색
2. **2묶음 후보 번들 생성:** 1묶음 번들 라이더의 교집합을 통해 2묶음 번들로 확장
3. **2묶음 번들 라이더별 최적 경로 탐색**
4. **penalty matrix 생성:** 2묶음 번들 생성 여부를 기반으로 주문 (i, j) 조합의 feasibility를 반영한 matrix 생성

1. 1묶음 번들 생성

1묶음 번들	1묶음 번들	
[1]	[2]	···
[Bike, Car, Walk]	[Car]	

- 용량 검사
- 용량 작은 라이더는 1묶음도 배달 불가능한 경우 존재

2. 2묶음 후보 번들 생성

1묶음 번들 [1] [Car] + 1묶음 번들 [2] [Car] = 2묶음 후보 번들 [1, 2] [Car]

- 1묶음 라이더의 교집합을 통해 2묶음 후보 번들 생성

3. 2묶음 번들 최적 경로 탐색

2묶음 후보 번들 [1, 2] [Car]

경로 탐색

		주문 [1, 2] 조합
feasible한 경로 존재	feasible !	penalty = 0
feasible한 경로 존재 X	infeasible !	penalty = 100

- 2묶음 후보 번들의 최적 경로 탐색
- 주문 (i, j) 조합에 대해서 어떤 라이더든 feasible한 경로가 존재하는 경우 -> penalty=0
- feasible한 경로가 존재하지 않는 경우 -> penalty=100 (묶일 수 없다.)

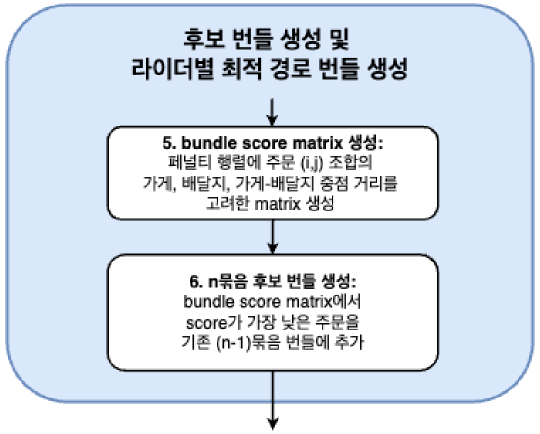
4. penalty matrix 생성

	0	100	...	0
0		0	...	0
100	0		...	100
⋮	⋮	⋮	⋮	⋮
0	0	100	...	

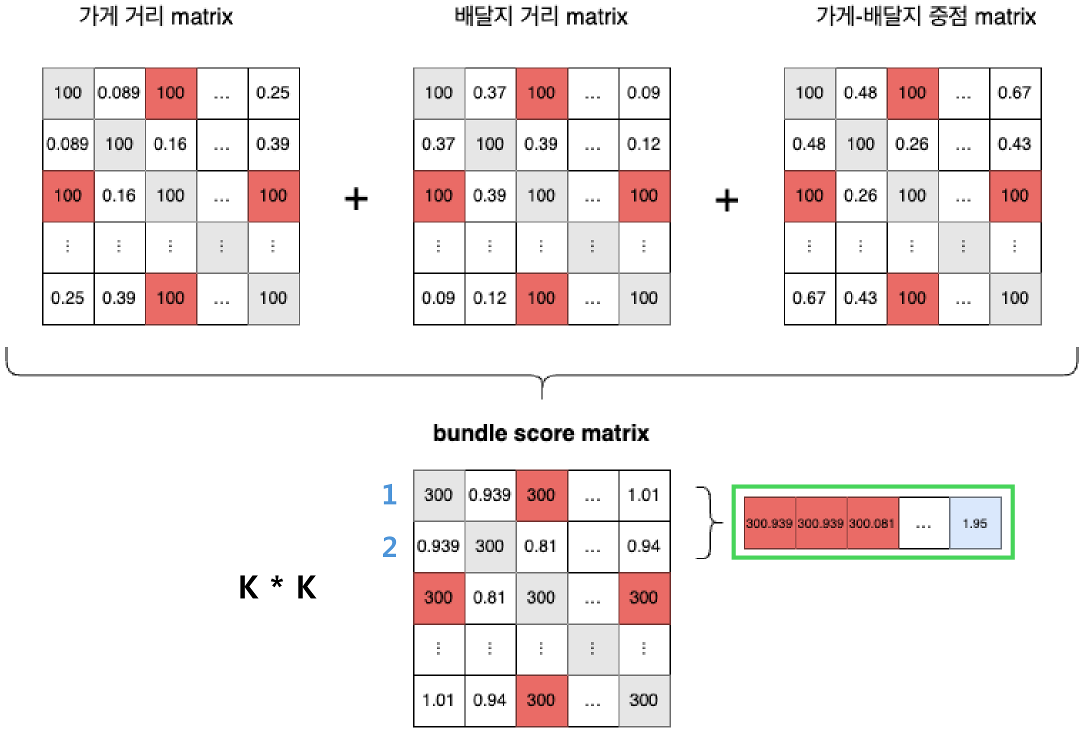
penalty matrix

K * K

- 모든 (i, j) 주문 조합에 대해 penalty matrix 생성
- 번들의 크기를 늘려가는 시도 중 불가능한 (i, j) 조합은 묶이지 않게 하기 위함



5. Bundle score matrix 생성, 6. n묶음 후보 번들 생성



① 주문 i, j 간의 가게 거리 표준화 행렬
② 주문 i, j 간의 배달지 거리 표준화 행렬
③ 주문 i, j 간의 가게-배달지 중점 거리 표준화 행렬

세 가지 행렬을 합쳐 bundle score matrix 생성

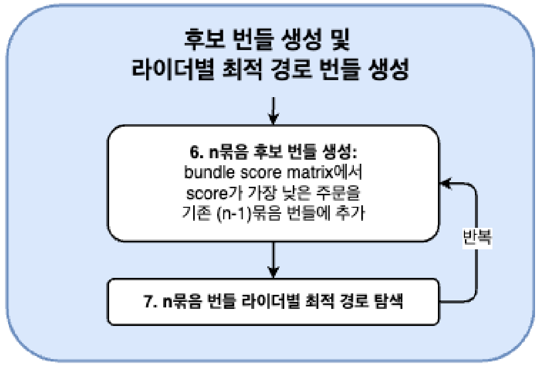
Why?

- 주문 간 Readytime 차이, deadline 차이 등 다른 요소들까지 고려한 로지스틱 회귀모형도 적용해보았으나,
- 단순하게 거리 행렬 세가지를 합친 score matrix의 결과가 가장 좋았음

- Bundle score matrix를 이용해 기존 (n-1)묶음 번들에서 n묶음 번들 후보 생성

예시)

- 주문 [1, 2] 2묶음 번들
- 3묶음 번들 만들 때 1행, 2행 합친 행 중 가장 낮은 score의 주문 추가



묶음 배송 제약

✓ **용량 제약**
✓ Deadline(DL) 이전에 배달되어야 하는 **시간 제약**

7. n묶음 번들 라이더별 최적 경로 탐색

계산량을 줄이기 위한 로직

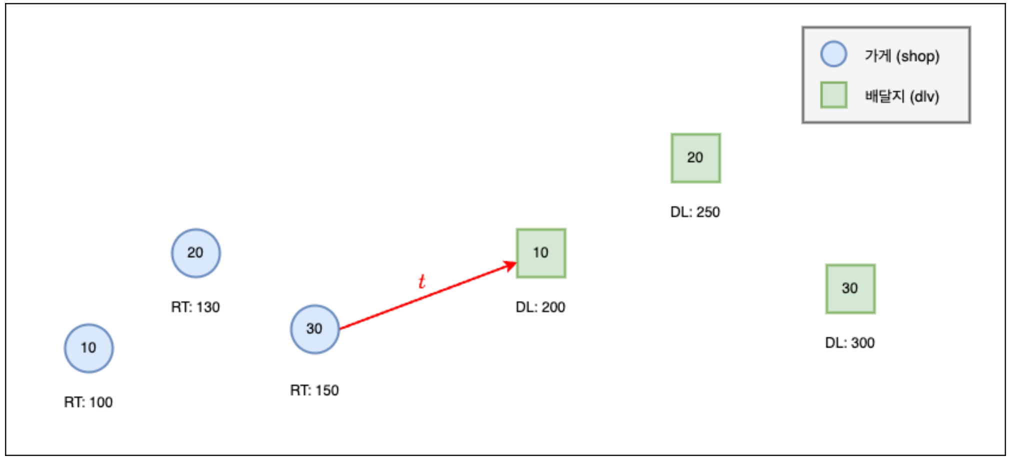
- 우선적으로 Infeasible 여부 판단할 수 있는 조건들

1) 주문 총 용량 > 라이더 용량

2) 가장 빠른 RT 주문 + 이동 시간 > 가장 빠른 DL 주문

3) 시간 제약 만족 여부를 판단하려면 마지막 shop seq에서 출발하는 시간만 계산하면 됨

예시) Orders: [10, 20, 30]



가장 빠른 RT 주문: [30] (RT: 150) , 가장 빠른 DL 주문: [10] (DL: 200)

150 + **t** > 200

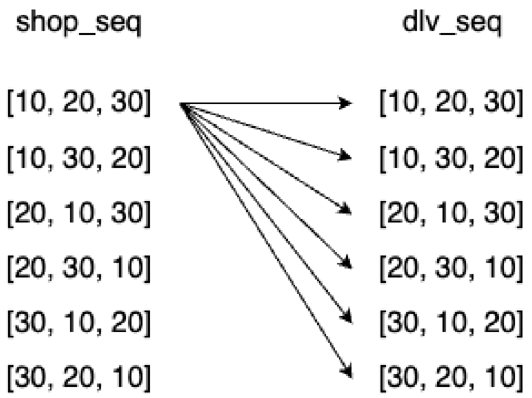
- **t** 계산해보면 빠르게 infeasible 여부 판단 가능

7. n묶음 번들 라이더별 최적 경로 탐색

계산량을 줄이기 위한 로직

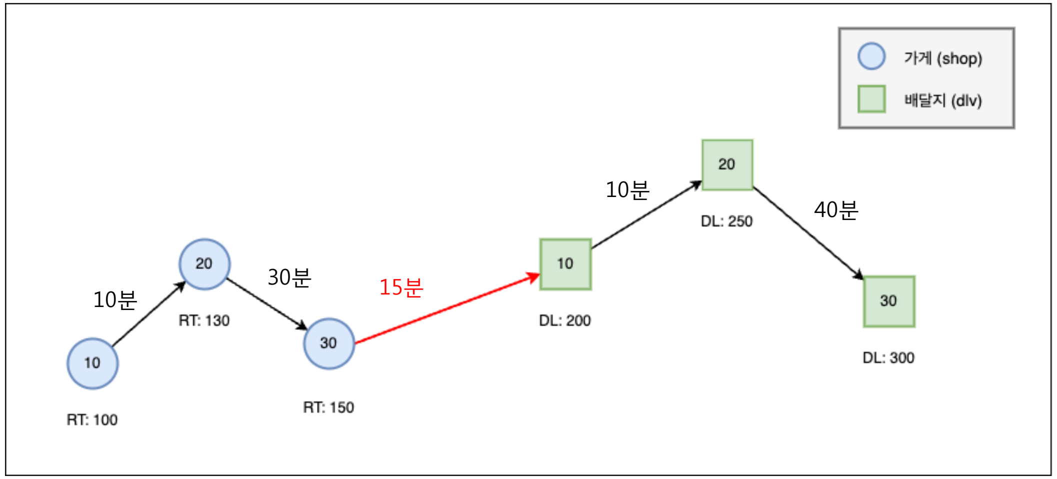
최적 경로를 구하기 위해선 permutation 계산이 필수적

- 중복되는 계산 多



- 각 shop_seq
  마지막 shop_seq에서 출발하는 시간, 누적 거리만 계산
- 각 dlv_seq
  dlv (i, j)간 누적 도착 시간, 거리 행렬 처음 1번만 계산한 뒤 저장

(예시) Orders: [10, 20, 30]



dlv_seq: [10, 20, 30]

거리 행렬: dlv_dist = [10~20 사이 거리, 20~30 사이 거리]
누적도착시간 행렬: dlv_arr_tbl = [0, 10, 10+40]

- dlv_seq의 모든 (i, j) 조합들에 대해 계산해두면
  shop_seq가 다르지만 dlv_seq가 같을 때,
  중복 계산할 필요 없이 가져다 사용 가능

7. n묶음 번들 라이더별 최적 경로 탐색

라이더별 최적 경로 구하는 로직

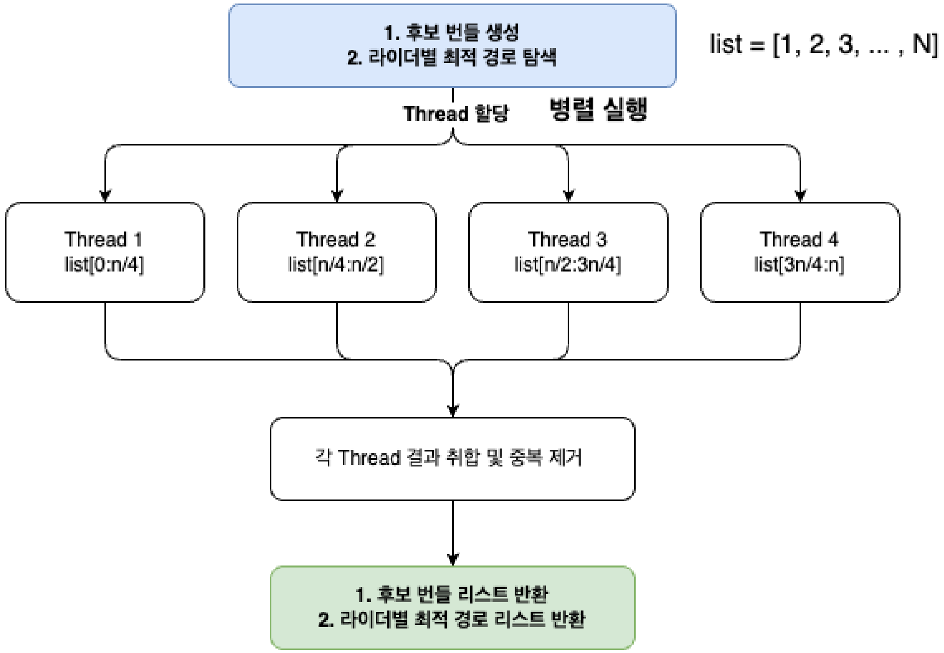
계산량을 줄이기 위한 로직

✓ 속도가 가장 빠른 라이더: Bike
✓ 용량이 가장 큰 라이더: Car

- Bike 속도로 못 가는 경로는 다른 라이더도 가능하지 않음
- Bike에서 용량 제약이 걸렸을 경우에 Car를 시도
- 모든 라이더에 대해서 경로 탐색 하지 않아도 됨

```
- Try Bike
    - If feasible:
        - Try Walk
        - Try Car
    - If infeasible by 용량 제약:
        - Try Car
```

시간 단축 시도



- 번들 생성 함수, 라이더별 최적 경로 탐색 함수 각각 4개의 Thread에 분할하여 병렬 계산
- 병렬 계산으로 시간 단축 시도

Sets & Parameters

- B : Set of feasible bundles
- K : Set of orders
- R : Set of riders
- A_r : Available number of rider r, $\forall r \in R$
- K_b : Set of orders belonging to bundle b, $\forall b \in B$
- R_b : Set of riders belonging to bundle b, $\forall b \in B$
- c_b^r : Total cost of bundle b performed by rider r, $\forall b \in B, r \in R$

Decision variables

- x_b^r : 1 if the bundle b is serviced by rider r, 0 otherwise, $\forall b \in B, r \in R$
- z_b^k : 1 if the order k is included in bundle b, 0 otherwise, $\forall b \in B, k \in K$

Set Covering
MIP formulation

$$\min \quad \sum_{b\in B} \sum_{r\in R_b} c_b^r \cdot x_b^r \; / \; |K|$$

$$\text{s.t.} \quad \sum_{b\in B} z_b^k \sum_{r\in R_b} x_b^r \geq 1, \quad \forall k \in K$$

$$\sum_{b\in B} x_b^r \leq A_r, \quad \forall r \in R$$

$$x_b^r \in \{0,1\}, \quad \forall r \in R, \forall b \in B$$

- 모든 주문을 덮을 수 있는(cover) 가능한 번들 조합을 찾는 Set Covering 문제로 정의
- **목적 함수**
  : 평균 배달 비용을 최소화하는 것
- 제약식
  1. 모든 주문은 번들에 1번 이상 할당되어야 한다.
  2. 종류 별로 라이더의 수는 한정되어 있다.

Set Covering 솔루션 중복 검사

(예시)

- 최종 솔루션 중복 발생 !

  ['BIKE', [**10**, 20], [20, **10**]]

  ['CAR', [**10**, 21, 31], [21, 31, **10**]]

  ⋮

- 중복 번들 부분집합 구하기

  ['BIKE', [10], [10]]
  ['BIKE', [20], [20]]
  ['BIKE', [10, 20], [20, 10]]

  ['CAR', [10], [10]]
  ['CAR', [10, 21], [21, 10]]
  ['CAR', [10, 31], [21, 10]]
  ['CAR', [21, 31], [21, 31]]
  ['CAR', [10, 21, 31], [21, 31, 10]]

Set Partitioning

$$\min \quad \sum_{b\in B} \sum_{r\in R_b} c_b^r \cdot x_b^r \; / \; |K|$$

$$\text{s.t.} \quad \sum_{b\in B} z_b^k \sum_{r\in R_b} x_b^r = 1, \quad \forall k \in K$$

$$\sum_{b\in B} x_b^r \le A_r, \quad \forall r \in R$$

$$x_b^r \in \{0,1\}, \quad \forall r \in R, \forall b \in B$$

- ➢ 최종 솔루션에 중복 번들 부분집합을 추가해 이 번들들로만 Set Partitioning 문제를 푼다
- ➢ 번들 수가 작아 빠른 시간에 솔루션 도출
- ➢ 각 주문이 정확히 한번만 할당됨
- ➢ 중복 할당 문제 해결 feasible한 솔루션 도출 보장

2. 알고리즘 구현

알고리즘 구현 언어

- **Python**

사용 패키지

- **Gurobi**: 최적화 솔버로, Set Covering 문제 해결에 사용
- **Numba**: Python에서 함수 실행 속도를 높이기 위해 JIT(Just-In-Time) 컴파일을 제공하는 패키지로, 알고리즘의 반복적인 계산 작업에 적용하여 실행 속도 개선에 사용

구현 기술

- 병렬 처리 및 성능 향상: Numba와 멀티 Thread를 활용하여 알고리즘 실행 속도 개선 시도
- 최적화 모델링: 문제를 수리 모형으로 정의하고, 최적화 Solver Gurobi로 해결

3. 알고리즘 특장점

알고리즘의 주요 특장점

1. **단순한 로직으로 파라미터 튜닝에 용이**: 알고리즘의 구조가 복잡하지 않아 파라미터 조정이 쉽다.
2. **문제 상황 변화에 유연한 대응**: 문제의 상황이 바뀔 때, 간단히 번들 크기를 조정하는 방식으로 새로운 조건에 맞춰 유연하게 대응할 수 있다.

(예시) timelimit가 짧다면, 최대 생성 번들 크기 파라미터만 바꿔 작은 크기의 번들로만 문제를 빠르게 풀게 할 수 있다.

4. 알고리즘 개선 방향

알고리즘 개선방향

1. 거리 및 시간 행렬의 더 효율적인 활용:

중복 계산을 피하기 위해 거리 행렬과 시간 행렬을 global하게 저장한다면, 이전에 계산한 행렬을 재활용할 수도 있다.

(예시) (n-1)묶음 번들에서 계산한 행렬을 n묶음 번들 경로 계산할 때 활용할 수도 있음.

하지만 번들 생성 부분이 단계적으로 엮여있어 복잡성이 증가할까봐 시도하지 못했다.

2. Column Generation 기법 적용:

문제 해결의 효율성을 높이기 위해 **Column Generation** 기법을 적용해 보고자 했지만, 시간적 제약으로 인해 시도하지 못했다.

5. 경진대회 참여 후기

경진대회 참여 후기

이번 경진대회를 통해 실제 문제 해결에 대한 깊은 통찰을 얻을 수 있었습니다.
특히, 제한된 시간과 자원 내에서 최적의 솔루션을 도출해야 하는 과정에서 많은 도전과 성취감을 느꼈습니다. 또한, 다양한 문제를 접하고 이를 해결하는 과정에서 기술적인 성장뿐만 아니라 문제 해결에 대한 새로운 접근 방식을 배울 수 있었습니다.

또한, 경진대회를 준비해주신 운영진분들께 깊은 감사의 말씀을 드립니다.
덕분에 소중한 경험을 할 수 있었고, 배움의 기회를 가질 수 있었습니다.

매년 정기적으로 경진대회가 열리면 좋겠다는 바람이 있습니다. 이를 통해 최적화 문제에 관심 있는 사람들이 지속적으로 도전하고 성장할 수 있는 기회가 제공되면 좋겠습니다.

Q&A