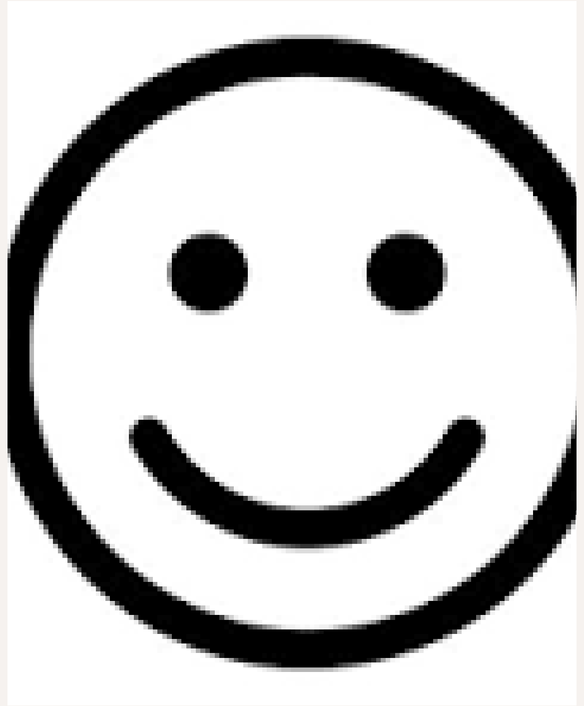
Bin packing problem

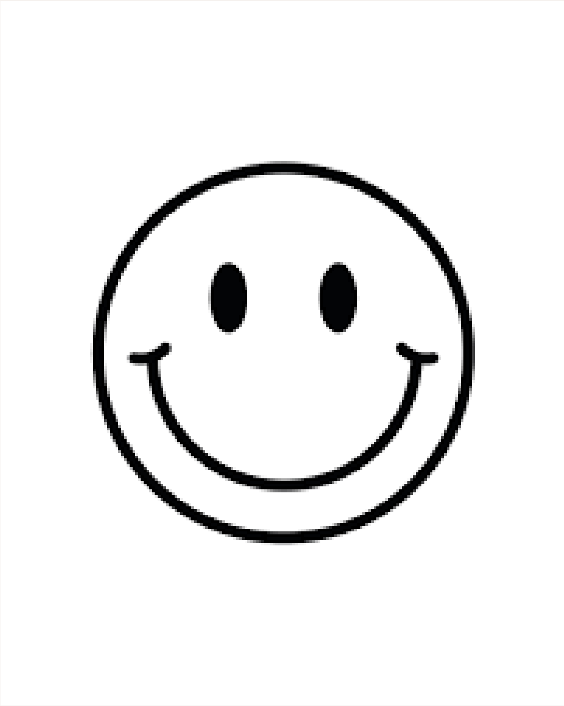
with Lower and Upper Bound Capacity Constraint

Class ID: 144217
Lecturer: Mr. Pham Quang Dung

OUR GROUP MEMBERS



Vu Hoang Nhat Anh



Phan Tran Viet Bach

Chu Trung Anh

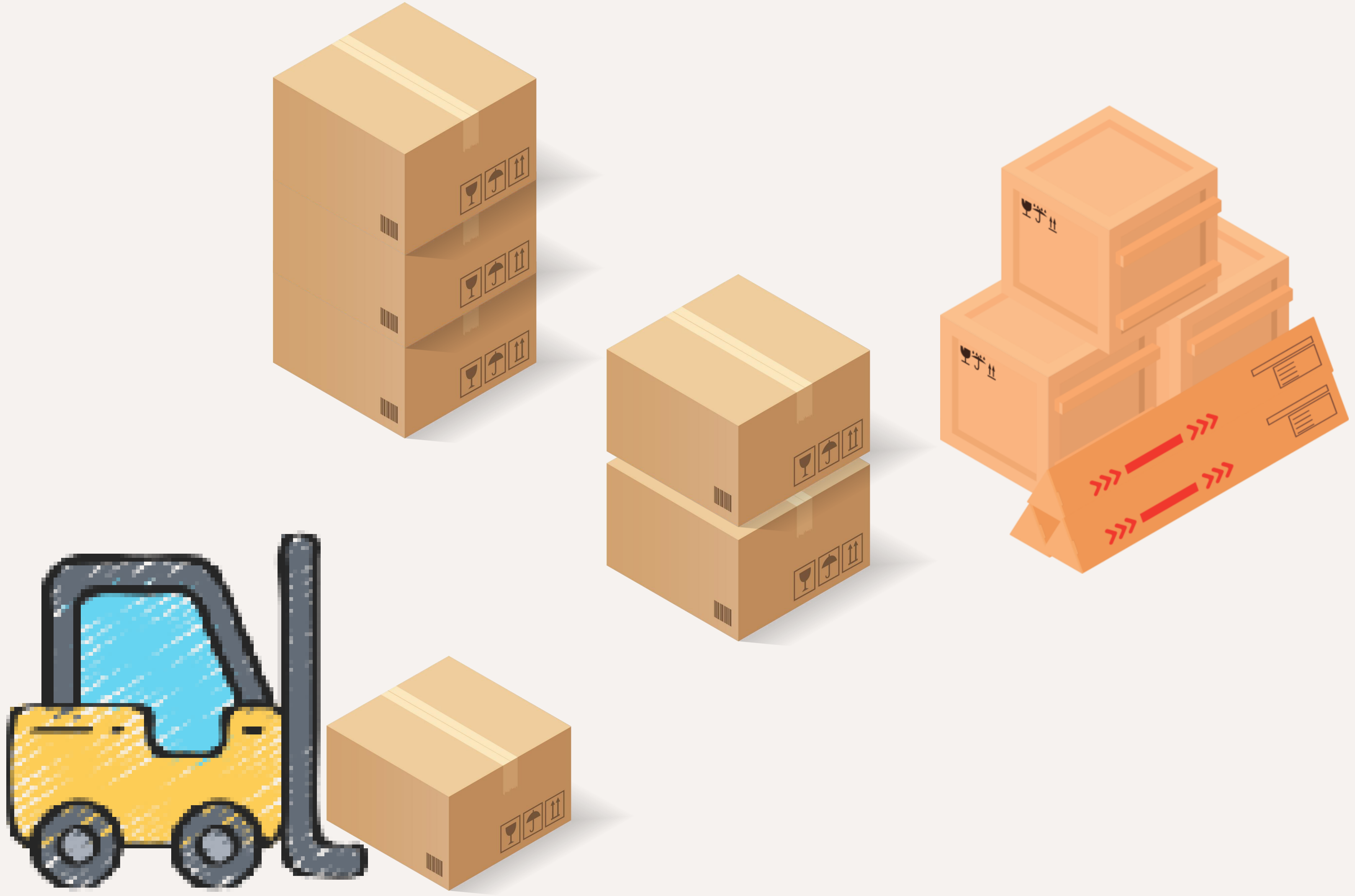
Tran Nam Tuan Vuong

Table of contents

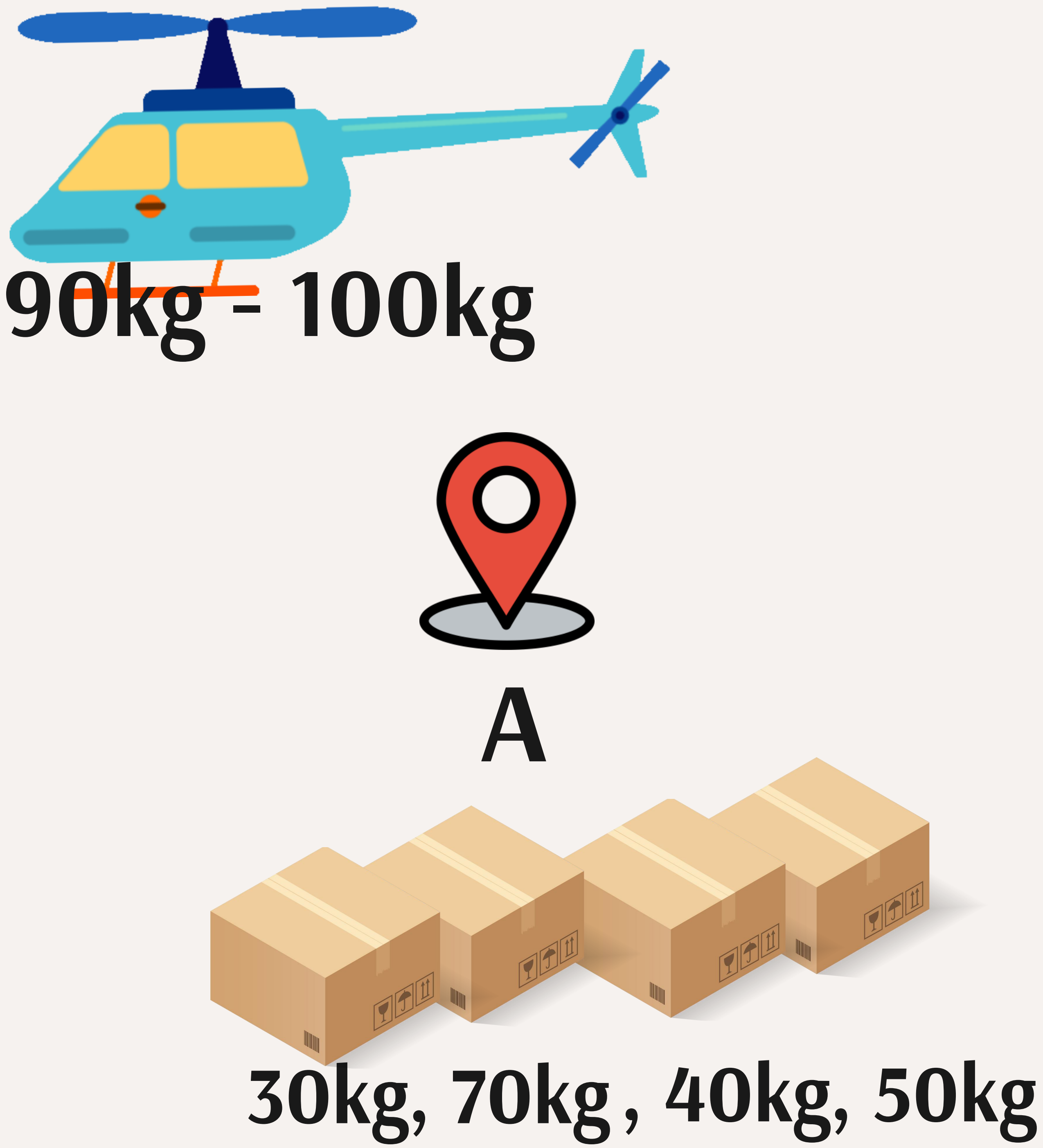
I. Problem description

The bin packing problem is an optimization problem in which items of different sizes must be packed into a finite number of bins or containers, each having a fixed given capacity, in a way that optimizes the problem depending on the desired outcome.

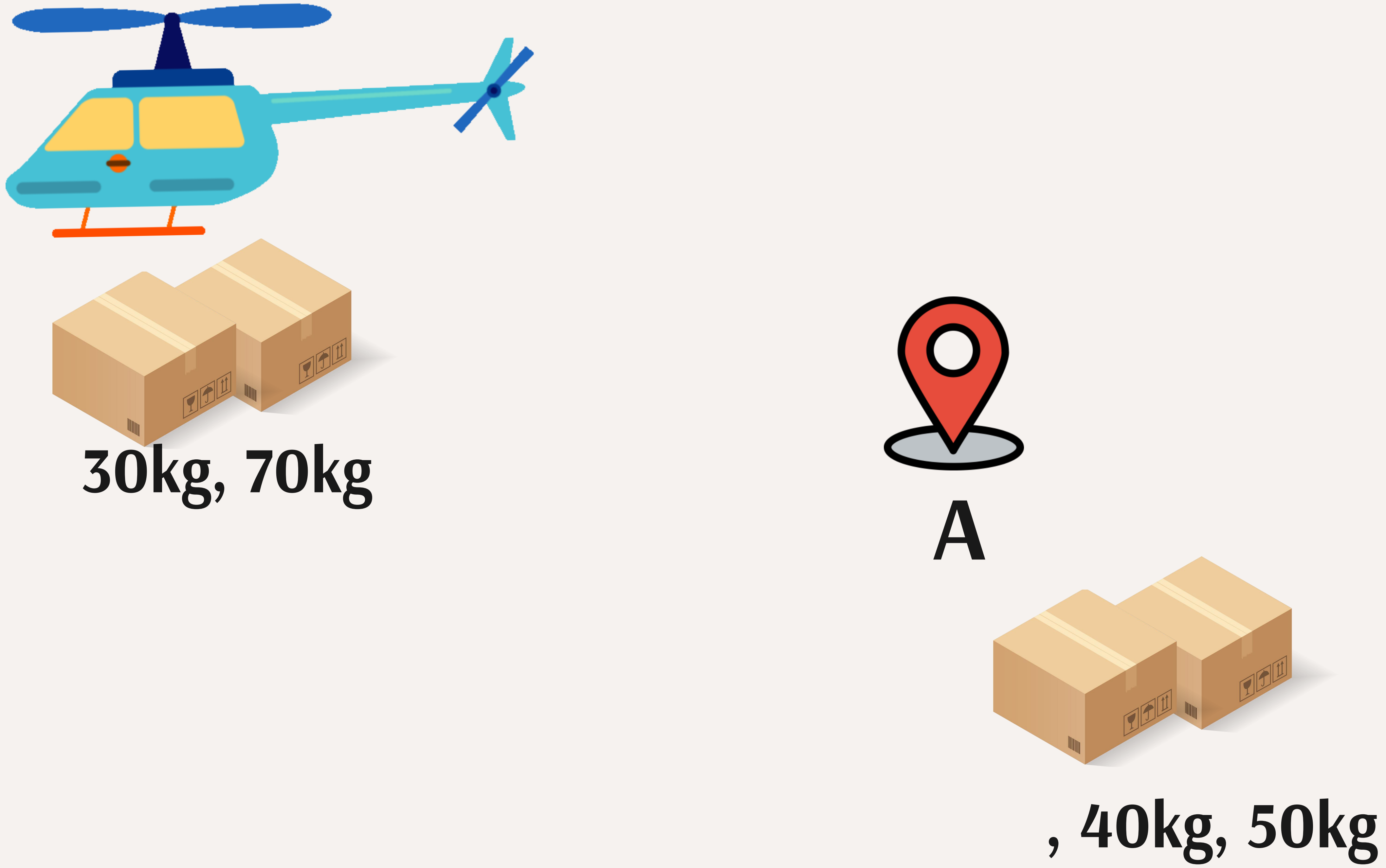
The goal in this case:
1) An “item” is packed to at most 1 “bin”;
2) Each “bin” must meet the requirements of loaded quantities: **a lower bound** & **an upper bound**;
3) The profit gained from “packed items” is maximal.



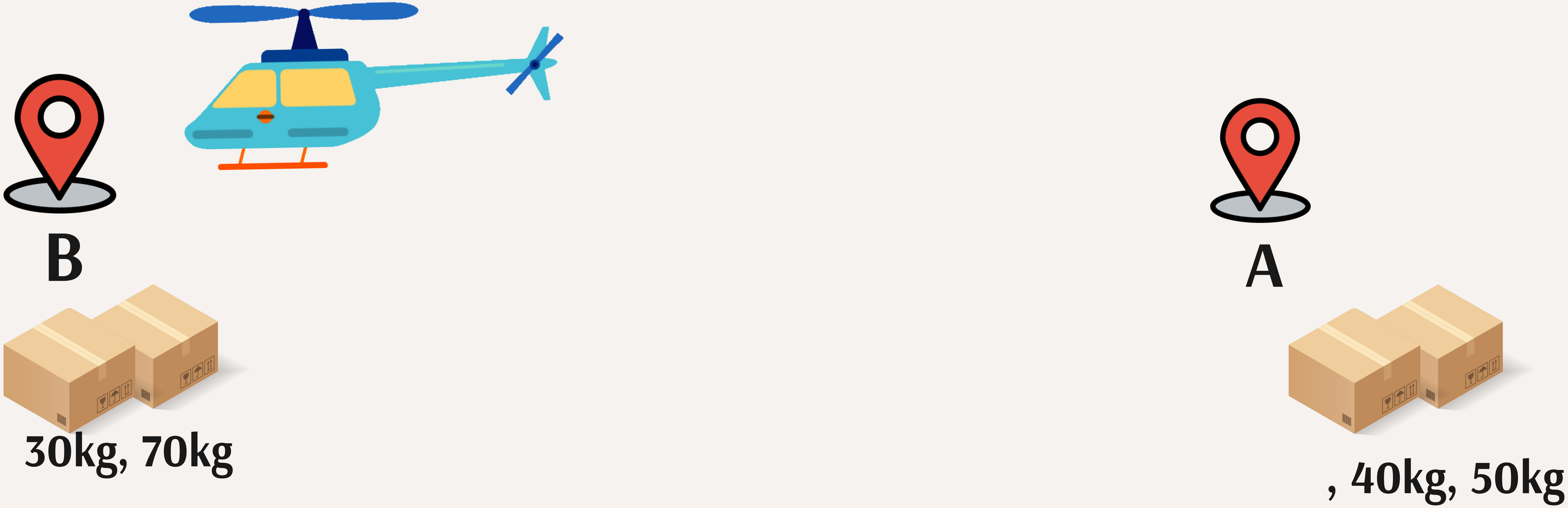
I. Problem description



I. Problem description



I. Problem description



I. Problem description

Data provided

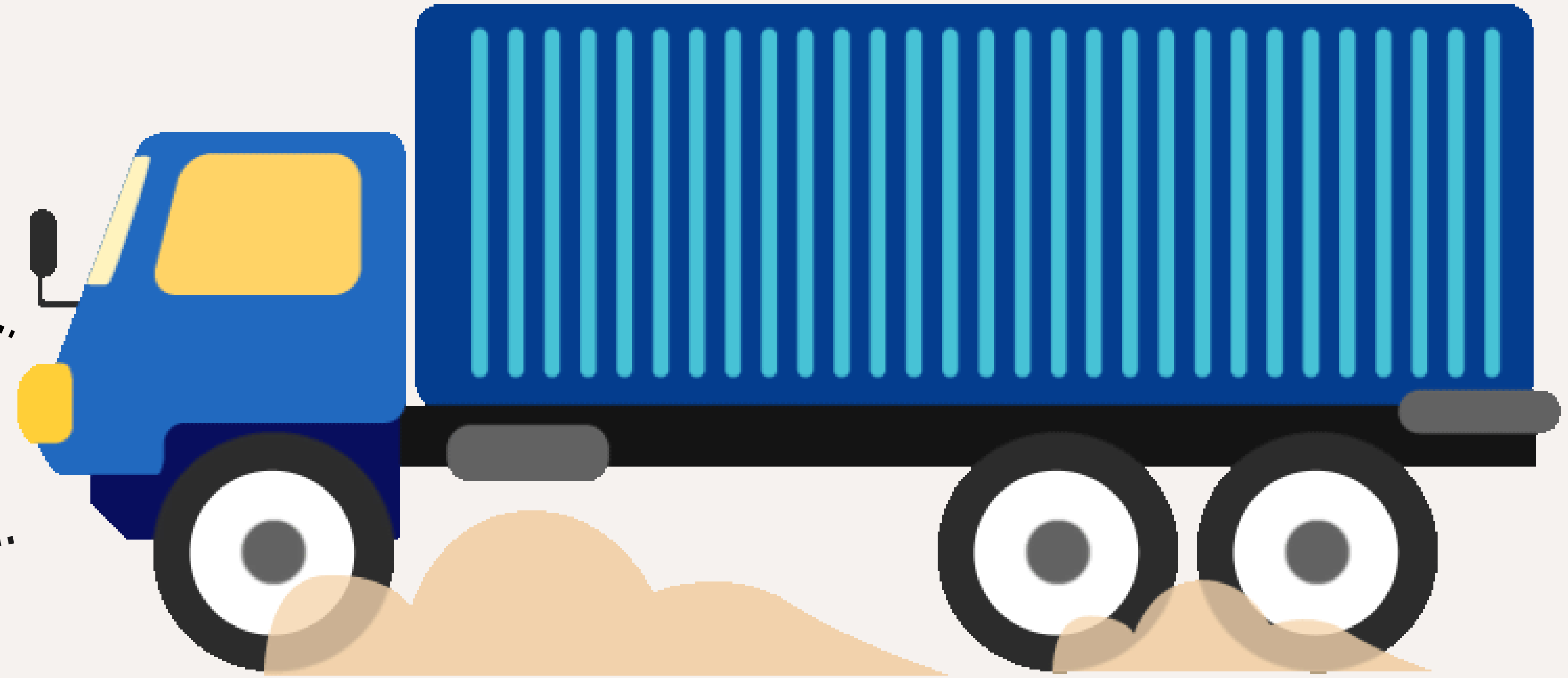
K
vehicles

N
orders

I. Problem description
Data provided

Lower bound **c1**

Upper bound **c2**



N
orders

I. Problem description
Data provided

K
vehicles

N
orders

I. Problem description
Data provided

K vehicles

Quantity (weight) **w**

Cost (profit) **p**

I. Problem description

Data provided

K vehicles ("bins")

Capacity bounds: lower bound c_1

and upper bound c_2

N orders ("items")

Quantity (weight) w

and cost (profit) p

I. Problem description

Modelling the problem

For $\boldsymbol{i^{th}}$ order, $\mathbf{1} \leq \boldsymbol{i} \leq \boldsymbol{N}$ and $\boldsymbol{j^{th}}$ vehicle, $\mathbf{1} \leq \boldsymbol{j} \leq \boldsymbol{K}$:

$c_1(j)$: lower bound for capacity of j^{th} vehicle
$c_2(j)$: upper bound for capacity of j^{th} vehicle

$p(i)$: profit of i^{th} order
w(i): weight of i^{th} order

Decision variable:

$X(i,j)$: a binary variable

$$X(i,j) = \begin{cases} 1, if\ i^{th}\ order\ is\ served\ by\ j^{th}\ vehicle \\ 0, otherwise \end{cases}$$

I. Problem description

Modelling the problem

Constraints

- Each order is served by at most one vehicle:

$$\forall\, i = 1, 2, \dots, N: \sum_{j=1}^{K} X(i,j) \leq 1$$

- Total weight of orders served by a vehicle must be between the low-bound and up-bound of capacity of that vehicle:

$$\forall\, j = 1, 2, \dots, K: \ c_1(j) \leq \sum_{i=1}^{N} X(i,j) \times w(i) \leq c_2(j)$$

I. Problem description
Modelling the problem

Objective

Maximize total profit of served orders:

$$\sum_{j=1}^{K}\sum_{i=1}^{N} X(i,j) \times p(i) \rightarrow max$$

II. Algorithms

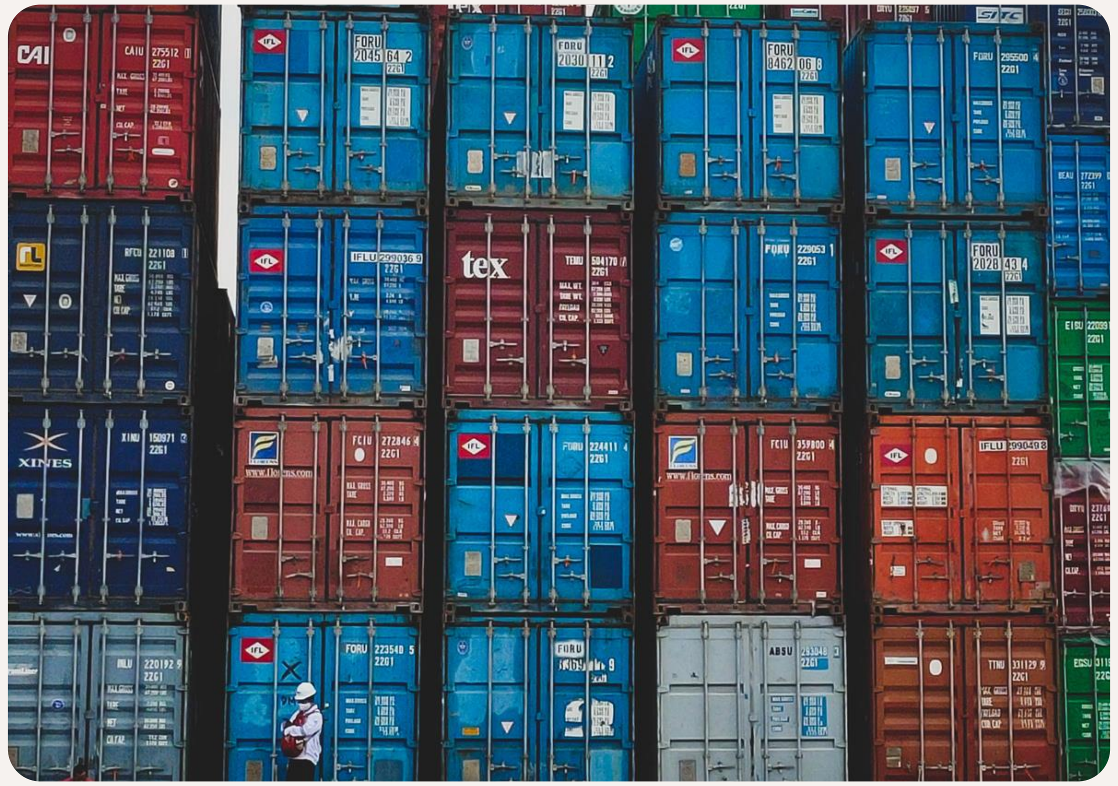
1. Solve exactly

- Integer Linear Programming (ILP)
- Constraint Programming (CP)

2. Handle larger cases

- “Greedy” assigning method
- Local search algorithm for completion

Algorithm 1
Integer Linear Programming



1. Integer Linear Programming

- Import the library

```
from ortools.linear_solver import pywraplp
```

- Get input data

```
n, k, Orders, Vehicles = GetInput()
```

- Create an ILP solver

```
solver = pywraplp.Solver.CreateSolver('SCIP')
```

1. Integer Linear Programming

- Get input data

```
def GetInput():
  n, k = map(int, input().split()) # n orders and k vehicles
  Orders = []
  Vehicles = []
  for i in range(n):
    w, p = map(int, input().split()) # weight and profit of
    Orders.append((w, p))            # each order
  for i in range(k):
    low, up = map(int, input().split()) # lower & upper bound
    Vehicles.append((low, up))          # for each vehicle
  return n, k, Orders, Vehicles
```

1. Integer Linear Programming

- Create decision variables

```
X = {}
for i in range(n):
  for j in range(k):
    X[i, j] = solver.IntVar(0, 1, X['+ str(i) + ',' + str(j) + ']')
```

$X(i,j)$: a binary variable

$$X(i,j) = \begin{cases} 1, if\ i^{th}\ order\ is\ served\ by\ j^{th}\ vehicle \\ 0, otherwise \end{cases}$$

1. Integer Linear Programming

- Add constraints

```
for i in range(n):
  c1 = solver.Constraint(0, 1)    # an order is not served
  for j in range(k):              # or served by 1 vehicle
    c1.SetCoefficient(X[i, j], 1)

for j in range(k):
  c2 = solver.Constraint(Vehicles[j][0], Vehicles[j][1])
  for i in range(n):                                   # total weight loaded
    c2.SetCoefficient(X[i, j], Orders[i][0]) # satisfies constraints
```

1. Integer Linear Programming

- Add objective function and call the solver

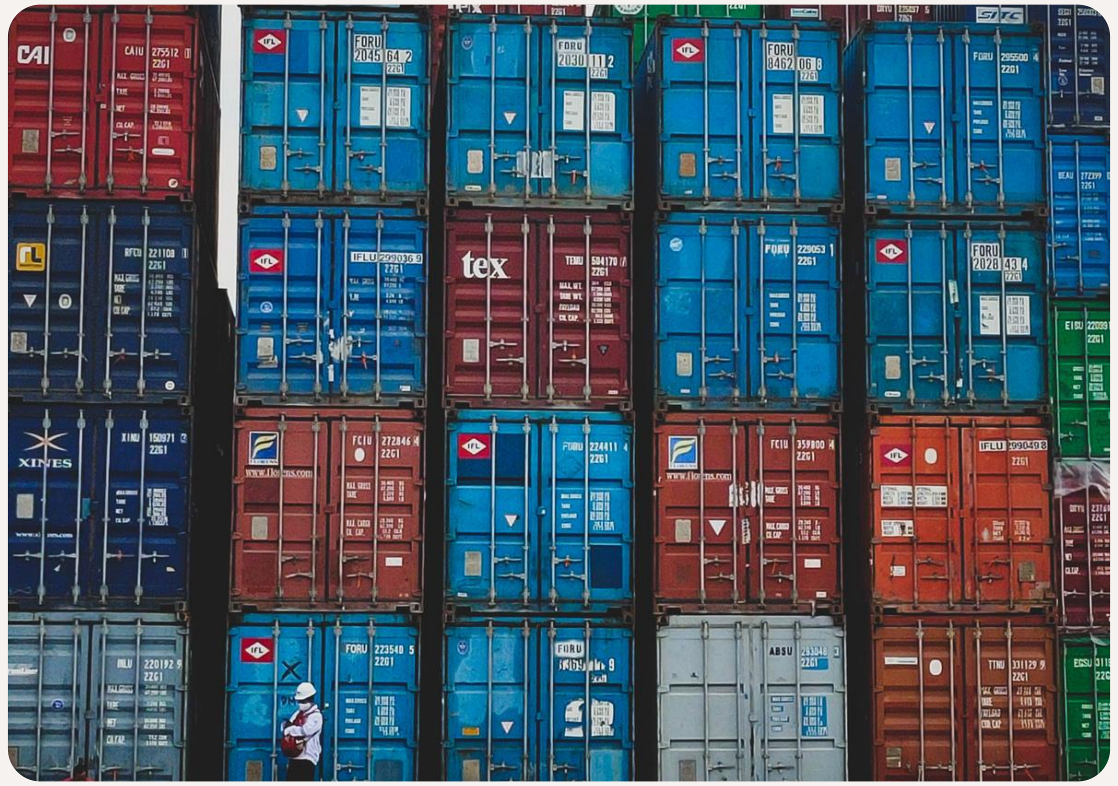
```
objective = solver.Objective()
for j in range(k):
  for i in range(n):
    objective.SetCoefficient(X[i, j], Orders[i][1])
objective.SetMaximization()
status = solver.Solve()
```

1. Integer Linear Programming

- Get results

```
if status == pywraplp.Solver.OPTIMAL:
  print(objective.Value()) # value of objective function
  order_count = 0          # counter of served orders
  solution = []            # pairs of (order, vehicle)
  for j in range(k):
    for i in range(n):
      if X[i, j].solution_value() == 1: # order i served
        order_count += 1                # by vehicle j
        solution.append((i+1, j+1))
  print(order_count)
  for order in solution:
    print(*order)
```

Algorithm 2
Constraint Programming



2. Constraint Programming

- Import the library

```python
from ortools.sat.python import cp_model
```

- Input data

```python
number_of_orders, number_of_vehicles = map(int,stdin.readline().split())

Orders = []
for i in range(number_of_orders):
    quantity, cost = map(int,stdin.readline().split())
    Orders.append(Order(i, quantity, cost))

Vehicles = []
for i in range(number_of_vehicles):
    low_capacity, up_capacity = map(int,stdin.readline().split())
    Vehicles.append(Vehicle(i, low_capacity, up_capacity))
```

2. Constraint Programming

- Class generation

```python
class Order:
   def __init__(self, order_id, quantity, cost):
       self.order_id = order_id
       self.quantity = quantity
       self.cost = cost
class Vehicle:
   def __init__(self, vehicle_id, low_capacity,up_capacity):
       self.vehicle_id = vehicle_id
       self.low_capacity = low_capacity
       self.up_capacity = up_capacity
       self.orders_vehicle_carry = []
       self.total_quantity = 0
```

2. Constraint Programming

- Model creation

```
model = cp_model.CpModel()
  x = {(i, j): model.NewBoolVar(f'x_{i}_{j}') for i in range(N)
       for j in range(K)}
```

- Constraints

```
for i in range(N):
    model.Add(sum(x[(i, j)] for j in range(K)) <= 1)

for j in range(K):
    quantity_sum = sum(orders[i].quantity * x[(i, j)] for i in range(N))
    model.Add(quantity_sum >= vehicles[j].low_capacity)
    model.Add(quantity_sum <= vehicles[j].up_capacity)
```

2. Constraint Programming

- Define the objective

```
model.Maximize(sum(orders[i].cost * x[(i, j)] for i in range(N) for j in range(K)))
```

- Create a solver

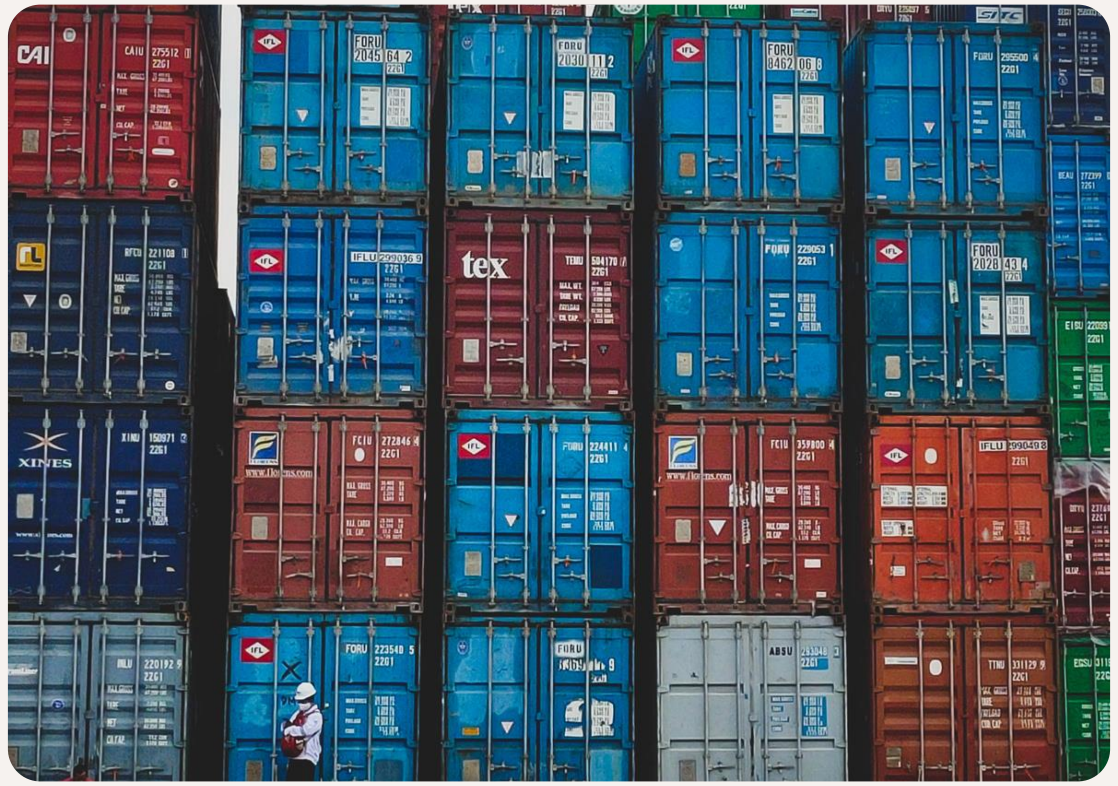
```
solver = cp_model.CpSolver()
status = solver.Solve(model)
```

2. Constraint Programming

- Solve the function

```python
solver = cp_model.CpSolver()
status = solver.Solve(model)

if status == cp_model.OPTIMAL:
    #print out the number of order were served
    m = 0
    for i in range(N):
        for j in range(K):
            if solver.Value(x[i,j]) > 0:
                m+=1
    stdout.write(str(m)+'\n')
    for i in range(N):
        for j in range(K):
            if solver.Value(x[i,j]) > 0:
                stdout.write(str(i+1) + ' ' + str(j+1)+'\n')
    stdout.write(f'Total cost: {solver.ObjectiveValue()}\n')
```

Algorithm 3
Greedy Algorithm



3. Greedy Algorithm

Random method

- Ignore constraints and randomly assign orders to vehicles
- Fast; creates an initial solution (for later use)

```
X = [-1 for i in range(N)]       # Decision variable
  for i in range(N):             # X[i] = j: order i served
    X[i] = random.randint(0,K-1) # by vehicle j
   # X[i] = -1 if order is not served
```

3. Greedy Algorithm

- Assign orders to vehicles sequentially to satisfy problem
- Sort orders and vehicles to support assigning process
- Easy constraint: always obey upper bound of capacity
- 2 approaches:
    - Satisfy capacity constraints (Greedy 1)
    - Maximize profit (Greedy 2)

3. Greedy Algorithm

Satisfy capacity constraints

- Satisfy as much capacity lower bounds as possible
- Strategy:
  - Sort vehicles decreasingly by capacity lower bounds
  - Sort orders decreasingly by weight
  - For each vehicle, check for orders in order list
  - If vehicle's low-bound is satisfied -> move to next vehicle

3. Greedy Algorithm

Satisfy capacity constraints

```
# n, k, Orders, Vehicles as in previous sections
# use enumerate() to mark indices of orders & vehicles
# sort by descending order
sorted_orders = sorted(enumerate(Orders), key=lambda x: x[1][0],
                       reverse=True) # order[0] := weight of order
sorted_vehicles = sorted(enumerate(Vehicles), key=lambda x: x[1][0],
                         reverse=True) # vehicle[0] := lower bound
```

3. Greedy Algorithm

Satisfy capacity constraints

We use 2 arrays to check for feasibility of orders and vehicles:

```
load = [0 for j in range(k)]             # load of vehicles
is_served = [False for i in range(n)] # an order is served or not
```

and an array to store results:

```
assignments = []
```

3. Greedy Algorithm

Satisfy capacity constraints

```
for j in range(k):
  vehicle_pos, cur_vehicle = sorted_vehicles[j]
  for i in range(n):
    if load[vehicle_pos] >= Vehicles[vehicle_pos][0]:
      break
  order_pos, cur_order = sorted_orders[i]
  if is_served[order_pos]:
    continue
```

3. Greedy Algorithm

Satisfy capacity constraints

```
if load[vehicle_pos] + cur_order[0] <= Vehicles[vehicle_pos][1]:
  is_served[order_pos] = True
  load[vehicle_pos] += cur_order[0]
  assignments.append((order_pos, vehicle_pos))
```

Results for unassigned orders:

```
for i in range(n):
  if not is_served[i]:
    assignments.append((i, -1)) # mark unassigned orders with -1
```

3. Greedy Algorithm

Satisfy capacity constraints

- Runs through each vehicle once, for each vehicle runs through all orders once -> O(n*k) -> fast
- Not feasible solution: Usually “smallest” vehicles are still underload
- Example: For a test case of 300 orders and 10 vehicles:

Vehicle 0: Load:789; lower bound:789
Vehicle 1: Load:1194; lower bound:1187
Vehicle 2: Load:778; lower bound:773
Vehicle 3: Load:1006; lower bound:1001
Vehicle 4: Load:694; lower bound:691
Vehicle 5: Load:585; lower bound:596
Vehicle 6: Load:622; lower bound:617
Vehicle 7: Load:728; lower bound:725
Vehicle 8: Load:938; lower bound:934
Vehicle 9: Load:886; lower bound:884

3. Greedy Algorithm

Maximize profit

- Prioritize orders with higher profit
- Strategy:
  - Sort vehicles increasingly by capacity upper bounds
  - Sort orders decreasingly by cost
  - For each order, attempt to load to a vehicle

3. Greedy Algorithm

Maximize profit

```
# Sort orders by cost in descending order
 sorted_orders = sorted(enumerate(orders, 1), key=lambda x: -x[1][1])

# Sort vehicles by upper capacity in ascending order
 sorted_vehicles = sorted(enumerate(vehicles, 1), key=lambda x: x[1][1])

 assignments = []
 remaining_capacity = [(capacity[0], capacity[1], i) for i, capacity in
                      sorted_vehicles]
```

3. Greedy Algorithm

Maximize profit

```
for order_count, (quantity, cost) in sorted_orders:
 assigned = False
 for i, (lower_bound, upper_bound, vehicle_count) in enumerate(remaining_capacity):
  if quantity <= upper_bound:
   assignments.append((order_count, vehicle_count))
   remaining_capacity[i] = (lower_bound-quantity, upper_bound-quantity, vehicle_count)
   assigned = True
   break
  if not assigned:
   assignments.append((order_count, 0)) # Mark as not served
```

3. Greedy Algorithm

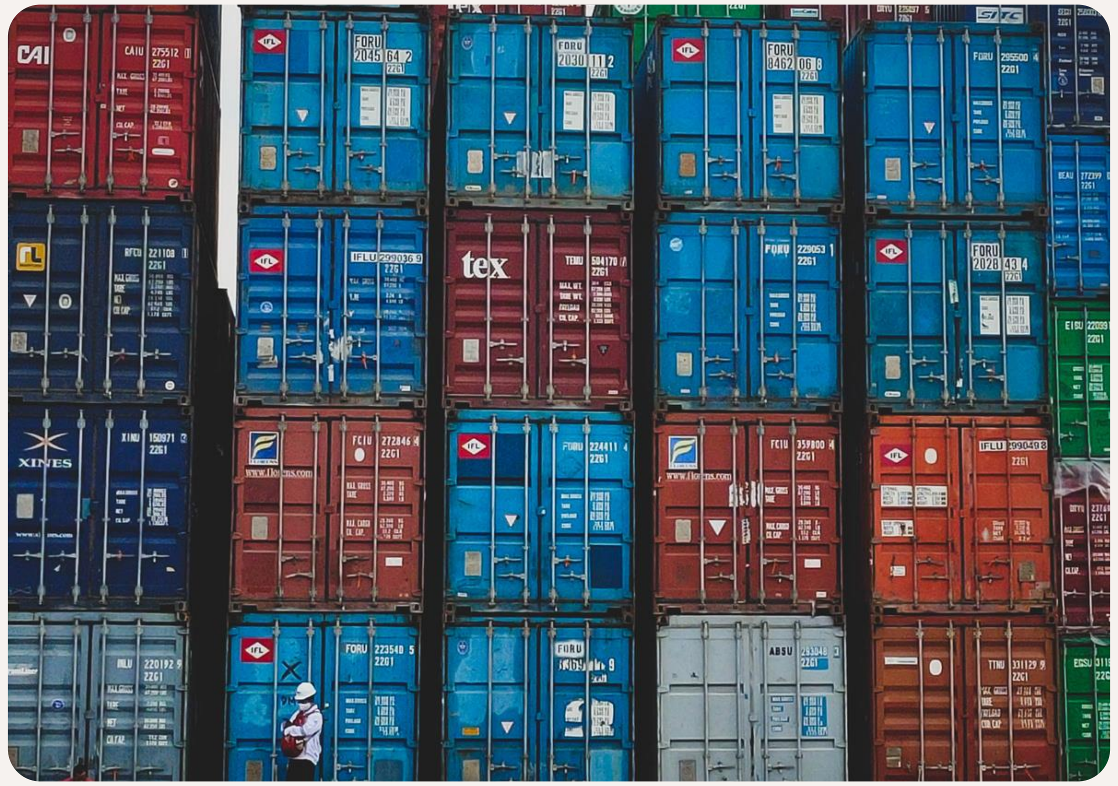
Maximize profit

- More optimal result compared to first strategy
- Still remains underload vehicle(s)

=> Require methods to complete the solution

=> Local Search strategy

Algorithm 4
Local Search Algorithm



4. Local Search

Initial Solution

1. Randomly assign orders to vehicles
2. Greedy algorithm (2)

Results of test cases 8, 9, 10

	Random algorithm	Greedy algorithm(2)
Time	29.1 sec.(total)	0.38 sec.(total)
Good move	742/969/1012	55/68/79
Bad move	48/62/65	2/3/5

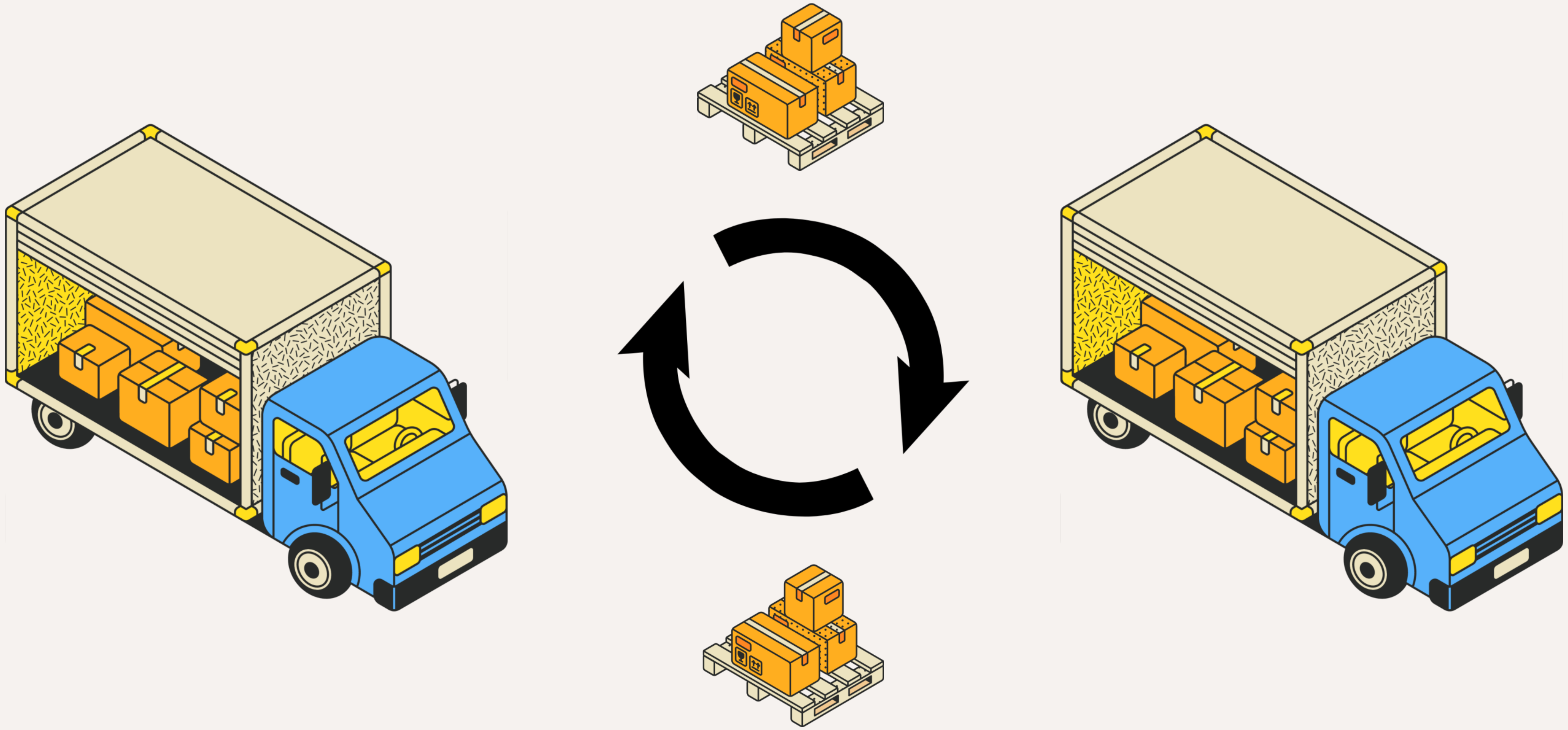
4. Local Search

Violation

```
violation = 0
[violation := violation + 1 for v in range(k) if\
sum_quant[v] > upper(v) or sum_quant[v] < lower(v)]
```

4. Local Search

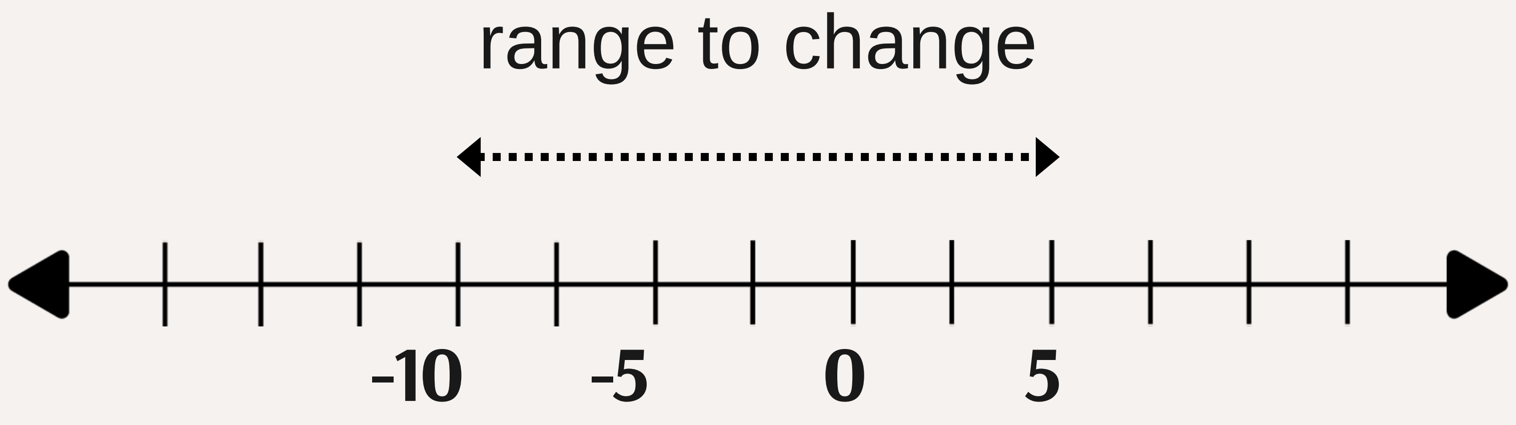
Idea of Local Move

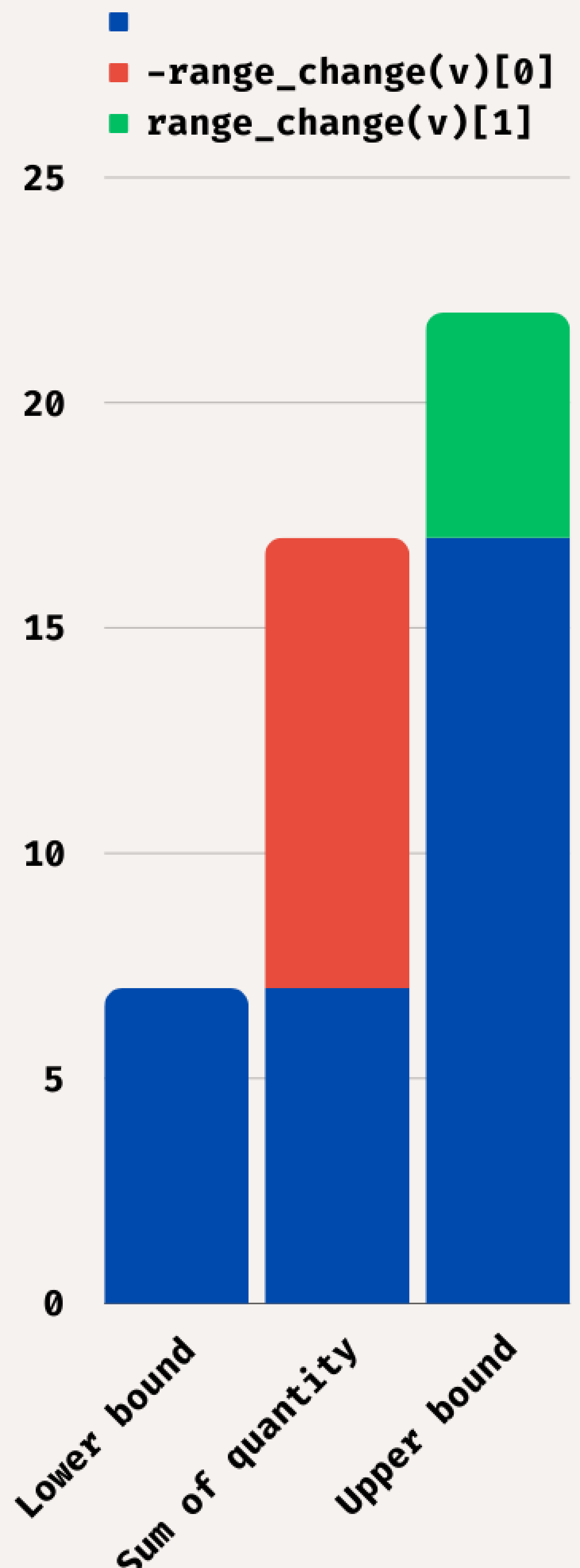


4. Local Search

Local Move Algorithm

```
def range_change(v):
    # The range that total load a vehicle can change
    return (lower(v) - sum_quant[v], upper(v) - sum_quant[v])

def can_go(v):
    #Check whether a vehicle can go: total load satisfied
    return range_change(v)[0] * range_change(v)[1] <= 0
```

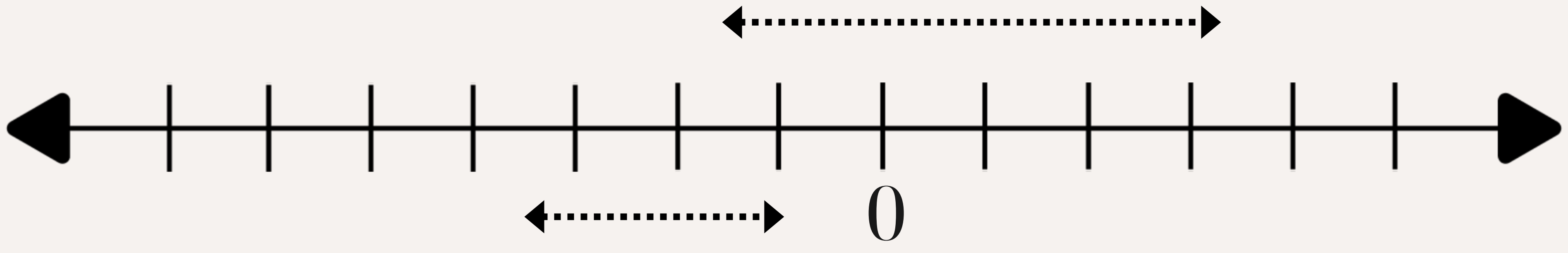




4. Local Search

Local Move Algorithm

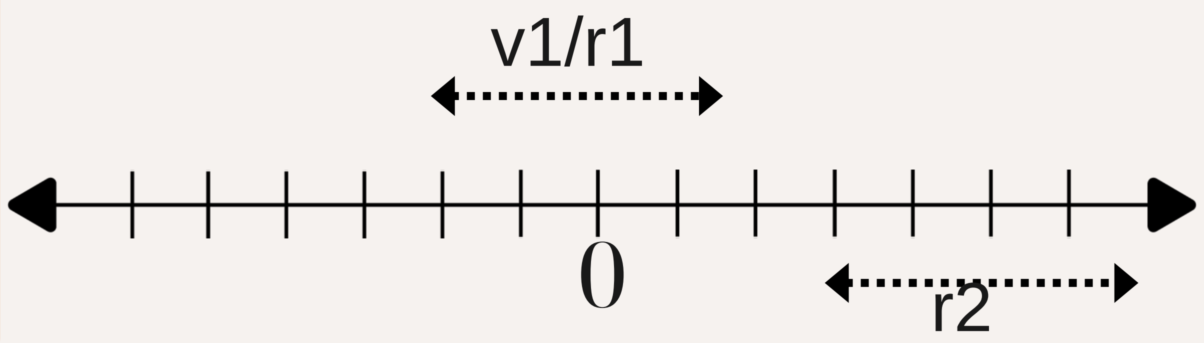
```
def can_trade(v1,v2):
    # Check whether 2 vehicles are suitable for the local move
    if can_go(v1) and can_go(v2):
        return False
    if range_change(v1)[0] * range_change(v2)[1] < 0 or\
    range_change(v1)[1] * range_change(v2)[0] < 0:
        return True
    return False

def choose_vehicles():
    # Randomly choose 2 vehicles which are suitable (can trade)
```

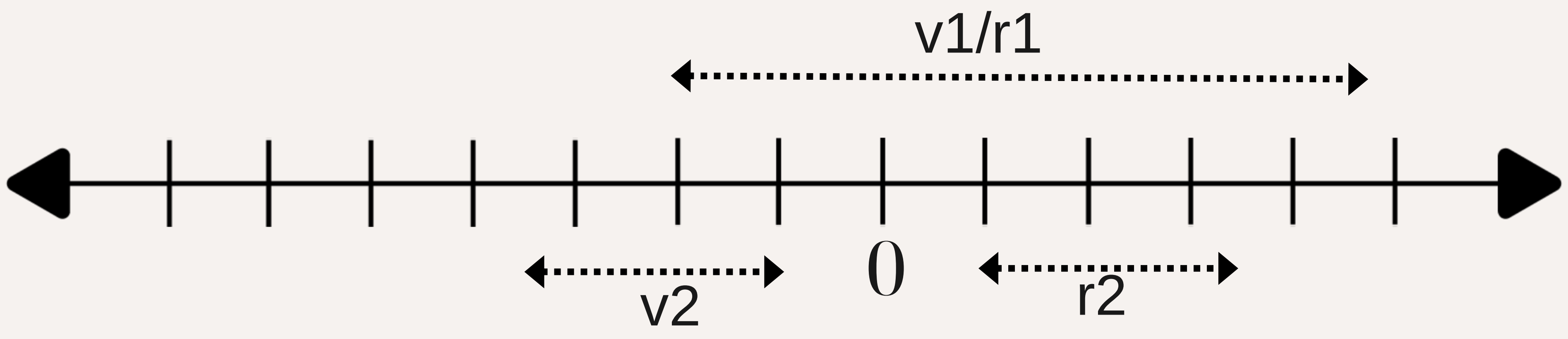


4. Local Search

Local Move Algorithm

```
def accept_change(v1,v2):
    # Acceptable range total quantity change
    # (+:v1,-:v2)
    r1 = range_change(v1)
    r2 = (-range_change(v2)[1], -range_change(v2)[0])
    if r1[0] < 0 and r2[0] < 0:
        ans = (max(r1[0], r2[0]), -1)
    elif r1[1] > 0 and r2[1] > 0:
        ans = (1, min(r1[1], r2[1]))
    return ans
```





4. Local Search

Local Move Algorithm

```
orders.append([0, 0]) # index-n
def choose_orders(v1, v2, accept_change):
  # Choose 2 sets of orders of each vehicle to trade
  candidate_v1 = [n]
  [candidate_v1.append(o) for o in range(n) if X[o] == v1]
  candidate_v2 = [n]
  [candidate_v2.append(o) for o in range(n) if X[o] == v2]
  for i in candidate_v1:
    for j in candidate_v2:
        if -quant(i) + quant(j) in\
        range(accept_change[0], accept_change[1]+1):
            return i, j, -quant(i)+quant(j)
```

4. Local Search

Propagation

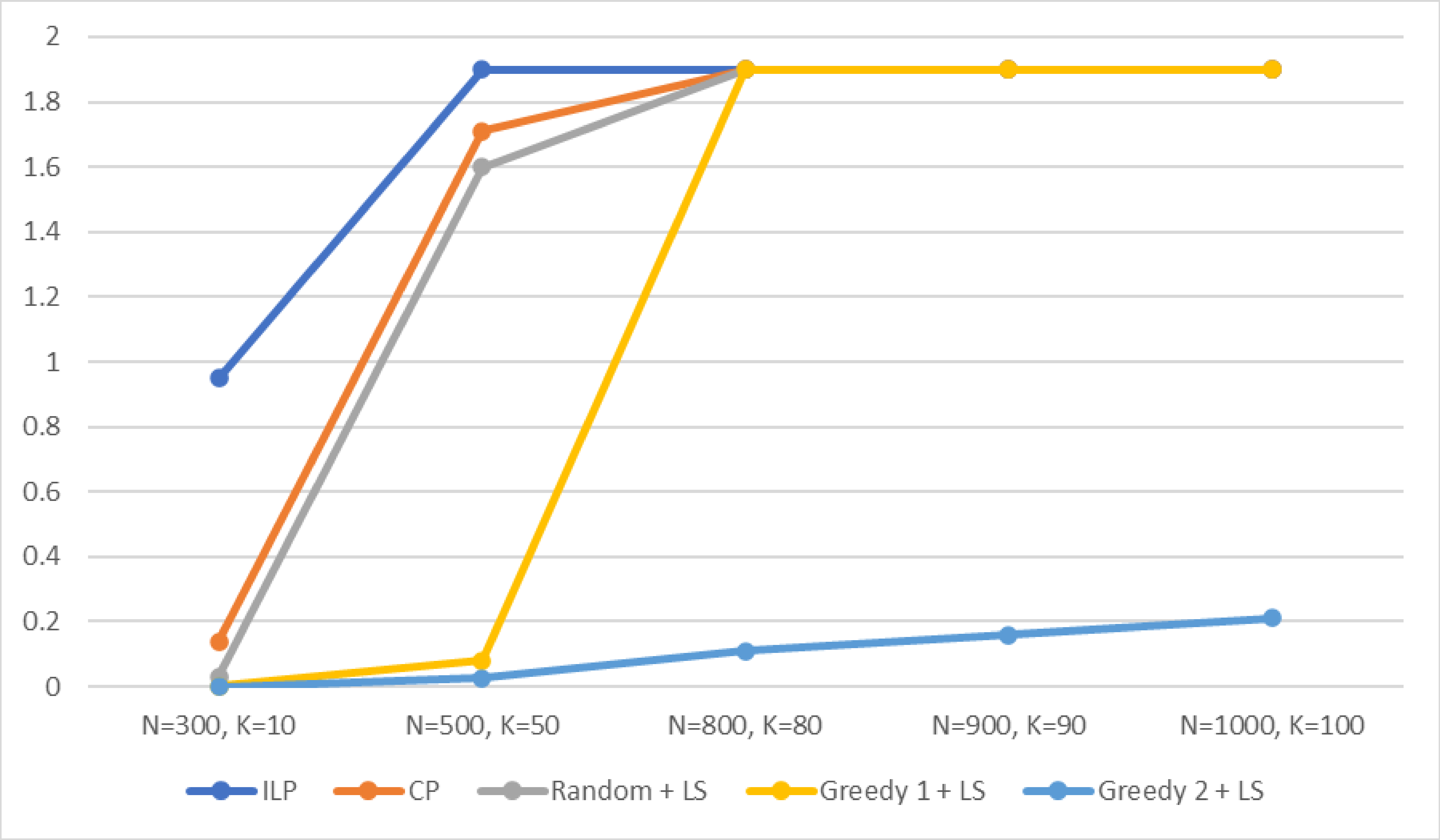
```
ans = choose_orders(v1, v2, ac)
if ans != None:
    pre_v1, pre_v2 = can_go(v1), can_go(v2)
    if ans[0] != n:
        X[ans[0]] = v2
    if ans[1] != n:
        X[ans[1]] = v1
    sum_quant[v1] += ans[2]
    sum_quant[v2] -= ans[2]
    if pre_v1 != can_go(v1):
        violation -= 1
    if pre_v2 != can_go(v2):
        violation -= 1
```

III. Tests and Results

- Results: Perfect
- Runtime: avg. of 10 tests, limit of 5 min.
  - For large test cases:

Test Cases	ILP	CP	Greedy + Local Search		
			Random	Greedy 1	Greedy 2
N=300, K=10	0.95	0.14	0.031	0.0025	6*10^-4
N=500, K=50	97.53	1.71	1.6	0.081	0.027
N=800, K=80	N/A	4.96	7.3	33.53	0.11
N=900, K=90	N/A	170	10.2	42.83	0.16
N=1000, K=100	N/A	9.48	11.6	51.69	0.21

III. Tests and Results

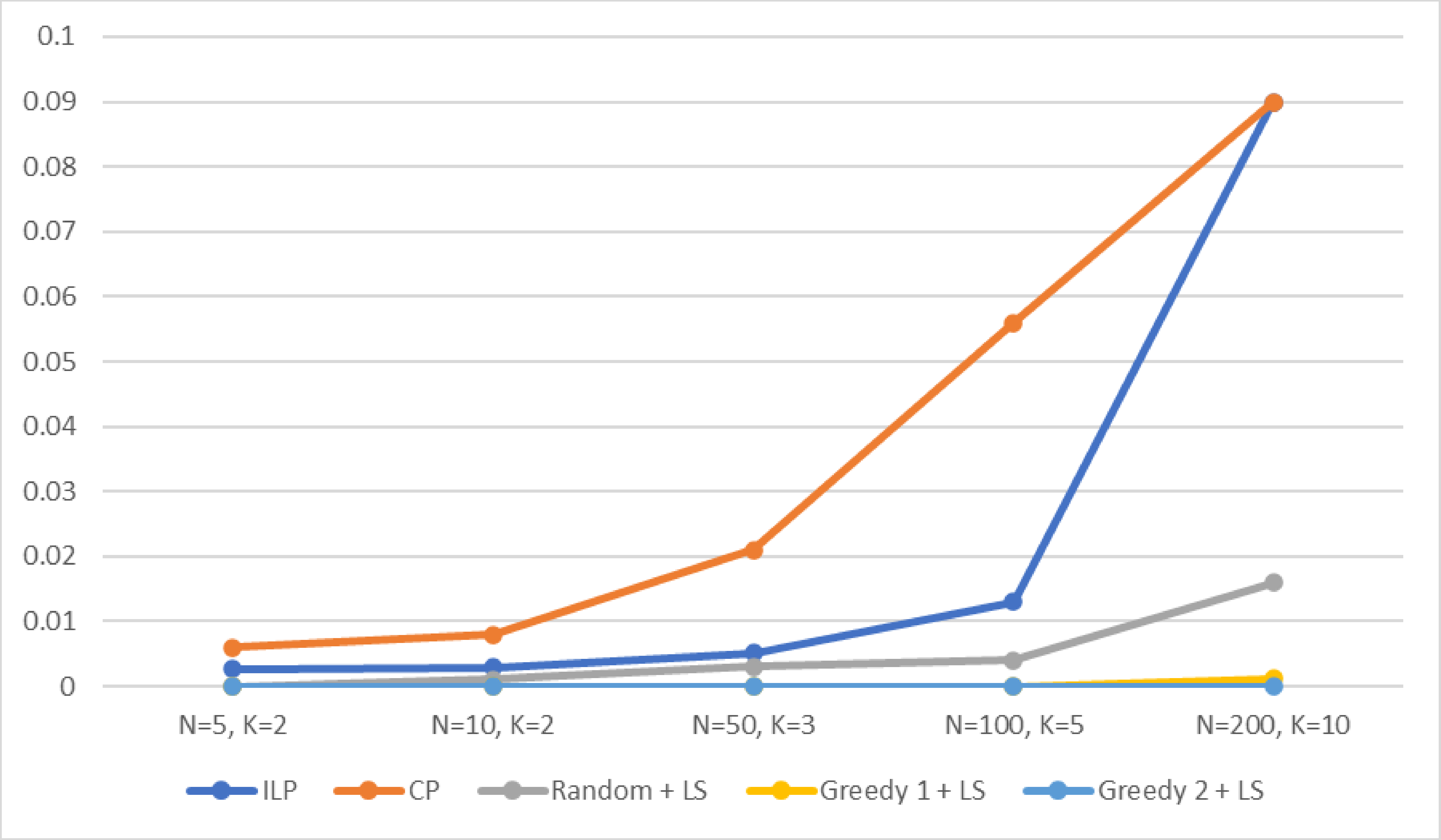


III. Tests and Results

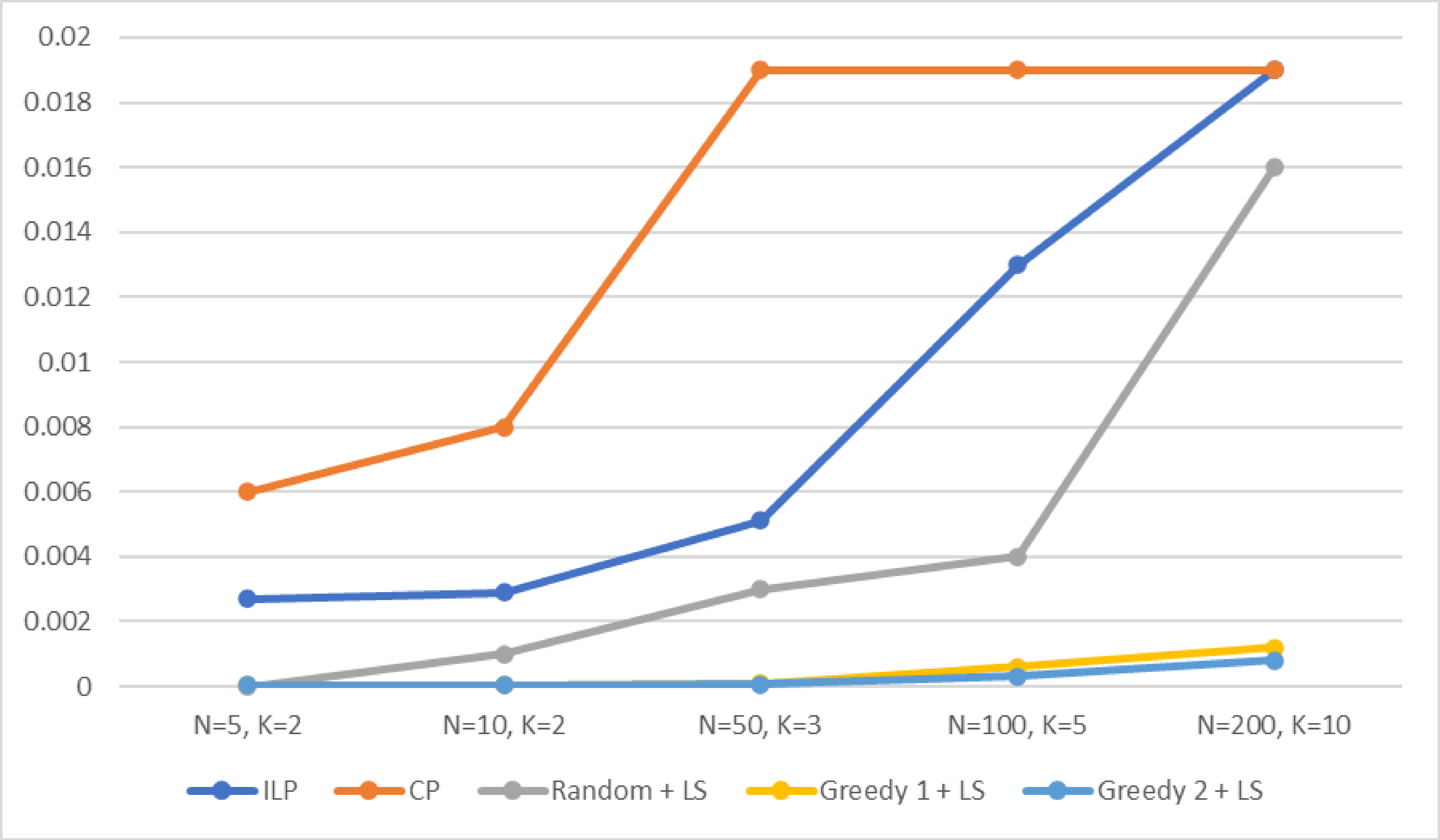
- For small test cases:

Test Cases	ILP	CP	Greedy + Local Search		
			Random	Greedy 1	Greedy 2
N=5, K=2	0.0027	0.006	10^-4	3 * 10^-5	3 * 10^-5
N=10, K=2	0.0029	0.008	0.001	4 * 10^-5	4 * 10^-5
N=50, K=3	0.0051	0.021	0.003	8 * 10^-5	7 * 10^-5
N=100, K=5	0.013	0.056	0.004	6 * 10^-4	3 * 10^-4
N=200, K=10	0.48	0.19	0.016	0.0012	8 * 10^-4

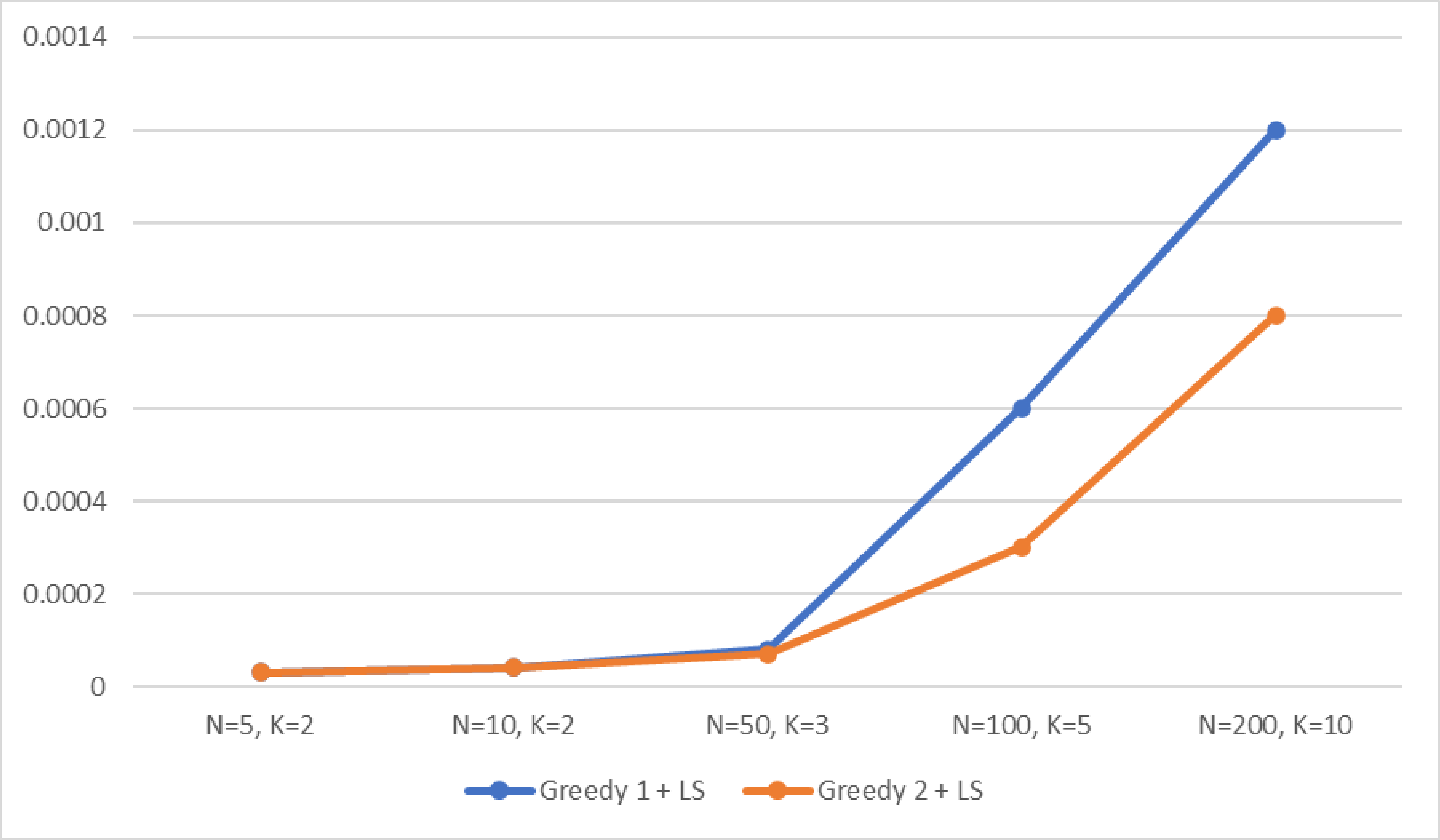
III. Tests and Results



III. Tests and Results



III. Tests and Results



III. Tests and Results

Conclusion

- Integer Linear Programming & Constraint Programming:
    - Provide exact results; long runtime, especially for large cases
    - For large cases, ILP has significantly longer runtime than CP

      => CP is more suitable for solving this problem
- Greedy algorithms with Local Search:
    - Greedy algorithms provide quick assignments; cannot satisfy all constraints
    - Local search: Faster than ILP & CP; give perfect results with given testcases.
    - Works best with “Greedy 2”: Prioritizing orders with higher profit & vehicles with smaller up-bound capacity

III. Tests and Results

Conclusion

- Drawbacks of local search:
    - Does not work when the difference between order quantites is too big.
    - Stop when all vehicles can go.
    - Not able to reduce sum of quantity of a vehicle whose sum of quantity is smaller than its lower bound.


THANK YOU FOR THE LISTENING
ANY QUESTION?
quickmeme.com