Link ppt:
https://husteduvn-my.sharepoint.com/:p:/g/personal/linh_tlm210535_sis_hust_edu_vn/EeQkmRHSk-xPhKx4lWzvD-UBi_SYztauiYrk7wVqtteGBw?e=OFJDSc
ờ chưa sửa đâu :v baoh cần xem thì lên
font code: Monaco, Menlo, Consolas, "Droid Sans Mono", "Inconsolata", "Courier New", monospace (fallback)

I. Problem description (Linh)

There exist ***N*** customers ***1, 2,..., N*** where customers order ***d[i]*** quantities of goods with the total value of ***c[i]***. There are ***K*** trucks ***1, 2,..., K*** transfering goods where truck k has a minimum load of c1[k] and maximum load of c2[k]. Compute the transferring plan satisfying:

- Each customer takes goods from only 1 truck.
- The total amount of goods on each truck must be in the range of the minimum and maximum loads of it.
- The total values of delivered goods must be maximized.

II. Modelling (Tuan)

- **Let**
    - ***goods_quantity = [d[1], d[2], d[3],..., d[N]]*** be the vector containing the quantity of goods that customers ordered
    - ***values = [c[1], c[2], c[3],..., d[N]]*** be the vector containing goods' values
- Load of each truck can be described as a weighted sum of goods' quantities where each weight can be either 0 or 1 indicating either that truck contains goods for that particular customer.

→ **Variables:**

weight[1:K, 1:N]: weight[i, j] is the boolean variable describing whether packages of customer ***j*** are contained on truck ***i***. Concretely:

- ***weight[i, j] == 1*** means that truck ***i*** contains packages of customer ***j***
- ***weight[i, j] != 1*** means that truck ***i*** doesn't contains packages of customer ***j***

- ***Constraints:***

- For each truck ***i***: load must be between lower bound and upper bound:

$$c1[i] <= weight[i, :]^T \times goods\_quantity <= c2[i]$$

- For each customer ***j***: his/her packages can only be delivered on one single truck:

$$sum(weight[i, j]) <= 1 \text{ for } i \text{ in } [1, K]$$

- ***Objective function to maximize***: Total values of delivered packages

$$sum(weight \times values)$$

III. Proposed methods

1. **Greedy (Tuan)**
   The first algorithm is very straightforward: we just put packages into trucks until all the constraints are satisfied. Concretely, we sort goods, and sequentially put them into trucks which are also sorted until the load of the truck that we're putting goods on meets its lower bound. At this point, we start putting goods on the next truck. We repeat until loads of all trucks meet their lower bounds. Next, we start putting the remaining packages (goods) into trucks until the trucks are full or there is no package left.

However, we have a problem: in which way should we sort packages and trucks? While with the trucks, sorting doesn’t really make any difference, packages sorting does. We propose 3 ways to sort them: by their quantities, values and by their efficiency.

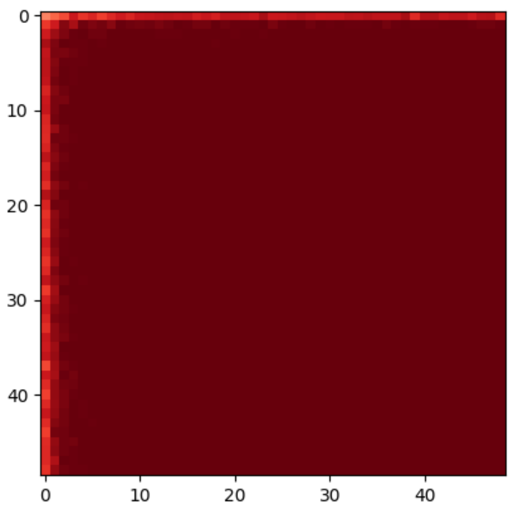
1.1. Sorting:

In this part, 2 different tests was used:

- The first test (A) is solve the problem with small values of N and K (N, K < 100)
- The second one (B) resembles reality much better with a very large value of N and much smaller value of K comparing to N: N = 100000 (10^5) and K = 200 ($2x10^2$)

1.1.1. Sort by efficiency

Efficiency of a package is defined as the rate of its value with respect to its quantity. In other words, efficiency = value / quantity. Delivering packages with higher efficiency gives us more values per quantity. In practice, this sorting method gives us the best total values. However, it has one major drawback: its failing rate. The failing rate measures the frequency that our method fail to give us a feasible solution. We measures this failing rate using test A and the result is shown below: Link ảnh



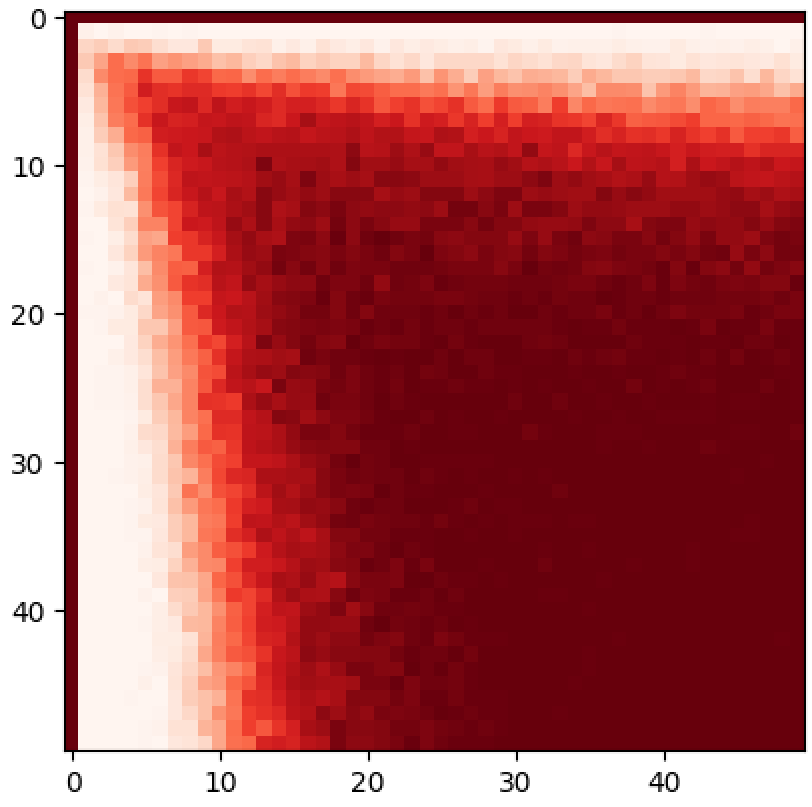
In this graph, the X axis corresponds to the value of K while the Y axis corresponds to the value of N. As being shown, most of the time, we don’t have a feasible solution. Test B also shows the same result: the solver is unable to give us a solution. We need either N or K to be small to have a change to have a solution. Otherwise, the algorithm fails.

1.1.2. Sort by value

As our objective function to be maximized is the total values of delivered packages, we try prioritizing the values of goods instead of efficiency. We ran test A and got about 90.54% goods delivered with total values 92.89% while the failing rate was a bit higher, especially when N is small: Link ảnh. When using test B, this approach almost never fail while giving us 99.62% of goods delivered with value rate of 99.75%

1.1.3. Sort by quantity

Here the only constraints that keep us away from a solution is the constraints about lower bound. With that in mind, we think that sorting by the quantity of goods might be a better bet. And in practice, it does. Look at the graph below [Link ảnh](), we can see that it gives us a much lower failing rates on test A. Moreover, in many cases, we have the failing rate falls to 0, which means our algorithm gives us a feasible solution every time.



Solutions from this approach in average give us about 93.97% goods delivered with values equal to 93.97% total values.

When tested with test B, sorting by quantity's value rate and delivered goods rate are both 97.59%.

1.1.4. Sort by a combination of value and quantity

As tested, sorting by the values gives us a higher value rate comparing to the deliver rate while they're equal when we sort using the quantities. We would want to combining both of them in the sorting process as ideally it would give us a higher value rate given the same deliver rate as sorting by quantities. Hence we try defining the importance of a package. First, we scale both values and quantities to a range of `[min_importance, 1]`, here `min_importance` (> 0) acts as a lower bound of importance as no package, even the one with lowest value should have an importance of 0. Next, we defined the importance of a package as below:

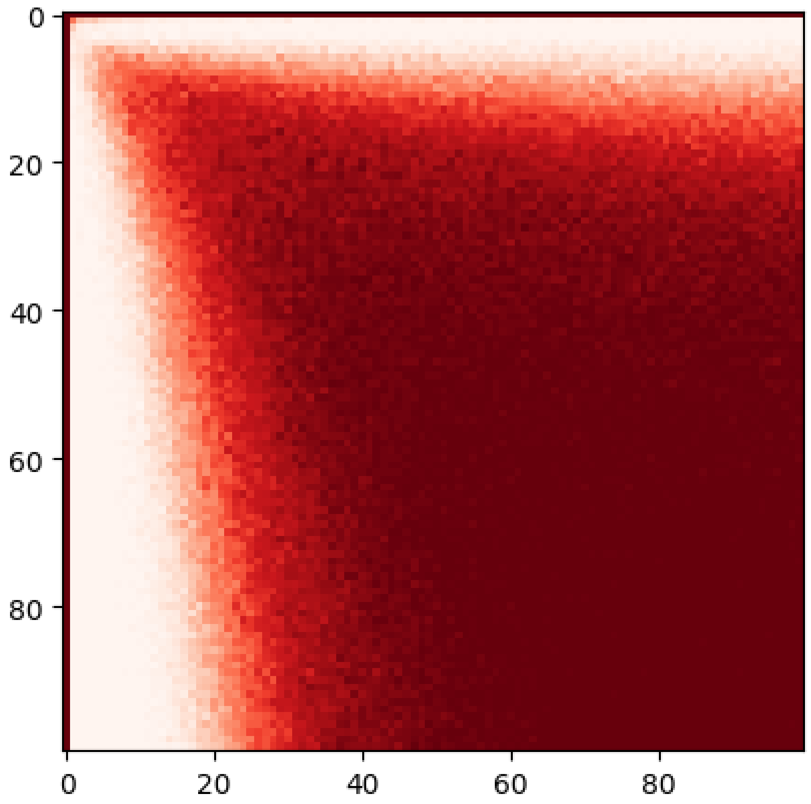
$$\texttt{importance} = \texttt{value}^{\texttt{order}} \times \texttt{quantity}$$

Here `order` is a parameter used to scale down the importance of `value` comparing to `quantity` as `value` is smaller than 1. In this way when sorting packages, we prioritize both values and quantity of packages with the values having lower priority. In our testing, we choose `order=2` as we didn't have enough time to tune this parameter. In practice, using this method, we get about 92.38% packages delivered with value rate of 94.17%, slightly better than sorting by quantity.

Test B shows that merging both method of sorting gives us an in-between result: 98.66% of values was delivered using 97.98% goods.

1.1.5. Combining different ways of sorting

After testing the aforementioned approaches, we use a combination of them: we sort those approaches by their rates of values delivered, try to use the one with the highest rate and check if it fails or not. If it fails, the next best method will be used. The order that we use is efficiency, value, importance and quantity.



[Link ảnh]

Hếc rùi ó

Method	Test A (N, K < 100)			Test B (N = 10^5, K = $2x10^2$)		
Result	Value rate (%)	Deliver rate (%)	Failing rate (%)	Value rate (%)	Deliver rate (%)	Failing rate (%)
Efficiency	88.38	86.57	93.62	0	0	100
Value	92.89	90.54	74.55	99.75	99.62	0

Importance	94.17	92.38	61.94	98.66	97.98	0
Quantity	93.97	93.97	29.69	97.59	97.59	0
Combine	96.78	96.05	29.26	99.75	99.62	0

1.2. Time complexity

- Python default sorting algorithm is Timsort (a combination of Merge sort and Insertion sort), which has a time complexity of O(n) in best case and O(nlogn) in worst and average cases. We use it to sort 2 lists with the same length: N → sorting time complexity: O(NlogN)
- Adding goods to trucks requires 2 nested for-loops, each has a time complexity of O(NK) → Adding goods time complexity: O(NK)

→ Time complexity: O(NK)

- In reality, greedy algorithm (if possible) instantly give us a solution in test A and takes less than a few seconds in test B.

2. Local Search

Greedy is a good algorithms for solving this problem with incredible speed
But fail rate is so high
=> Using greedy to find a solution having the high value of objective function which could not satisfy all constraints, then find its neighbour which satisfies all constraints to set initial solution for local search algorithm

2. 1. Hill climbing
Hill climbing is the first method used in local search. Starting with sufficient point then find around itself the better solution until reach the time limit.

```
Hill_climbing(timelimit)

      greedy_solution = Greedy()
      initial_solution= create_neighbour(greedy_solution)
      current_solution = initial_solution
      current_score = objective_function(initial_solution)
      best_solution = current_solution
      best_score = current_score

      while True:
            if time >= timelimit:
                  return best_score, current_solution

            next_solution = create_neighbour(current_solution)
            next_score = objective_function(next_solution)
```

```
        if best_score <= next_score:
            best_solution = next_solution
            best_score = next_score
        else:
            current_solution = next_solution
```

2. 2. Simulated annealing

Usually, Hill Climbing is stuck in local maximum, plateau, or shoulder. In order to jumping out this, Simulated annealing is a better algorithm to find global maximum
As recommended, temperature, cooling_rate, timelimit is 1000, 0.7, 60s respectively in default. This method chooses neighbour randomly at each temperature, and accept it with a probability p = $e^{\frac{\Delta E}{k.T}}$
In this case, k is average of all delta E of temperature reduction. If this method reach time limit, it returns the best feasible solution.

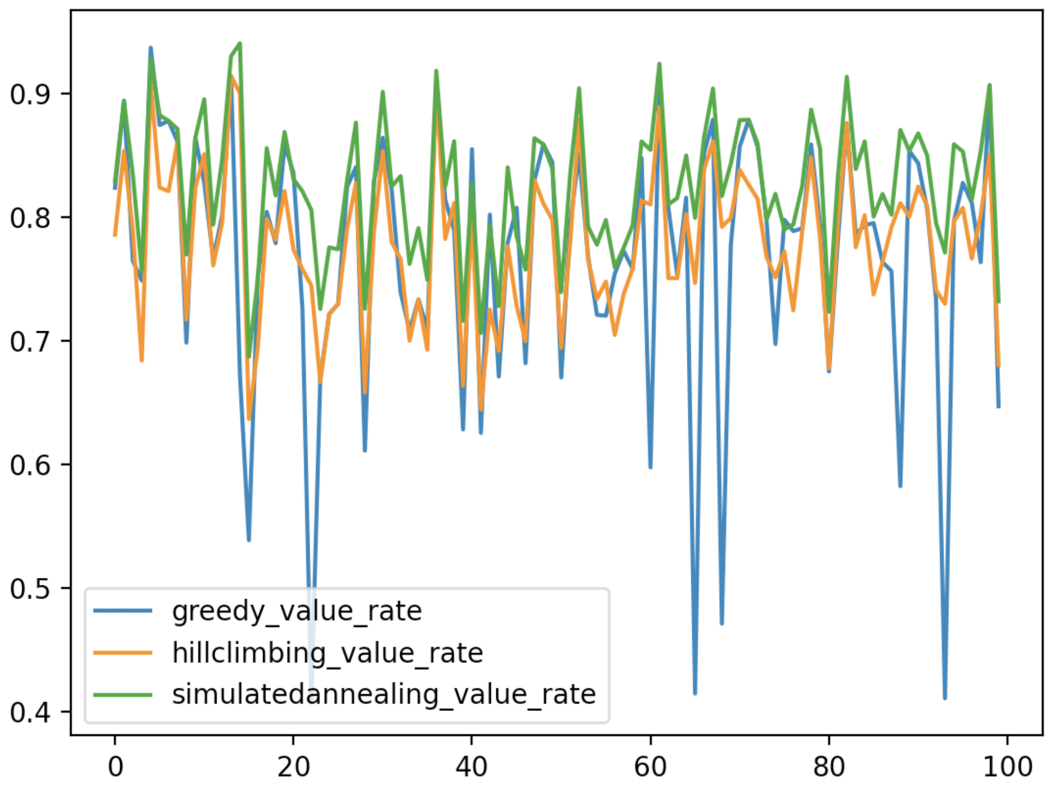
```
Simulated_annealing(temperature, cooling_rate, timelimit)

    current_temperature = temperature
    greedy_solution = Greedy()
    initial_solution= create_neighbour(greedy_solution)
    current_solution = initial_solution
    current_score = objective_function(initial_solution)
    best_solution = current_solution
    best_score = current_score
    n = 1

    while True:
        for t = 1 to inf:
            if time >= timelimit:
                return best_solution, best_score

            next_solution = create_neighbour(current_solution)
            next_score = objective_function(current_solution)
            ΔE = current_score - current_score
            k = abs(ΔE)
            if ΔE < 0:
                p = $e^{\frac{\Delta E}{k.T}}$
                if random() < p:
                    accept = False
        else:
                accept = True
else:
        accept = True
if accept = True:
        current_score = next_score
        current_solution = next_solution
        n +=1
        k = (k*(n-1) + ∆E) / n
current_temperature *= cooling_rate

if best_score <= current_score:
        best_solution = current_solution
        best_score = current_score
```



Quick Test: N=50, K=2

3. **Ortools ILP (Truong)**
   - .The Strengths and Weaknesses of this method:
     + Strengths: This method gives us quite fast and accurate result when the number of trucks and customers is small. The only job is to define the variables and its constraints, the rest is for the library.

    + Weaknesses: For large datasets, that means OR-Tools takes a long time to run. We can find many other libraries or other method to support.

- Import the libraries

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

- Creat the data

- Declare the MIP solver

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

  The default OR-Tools Mixed Integer Programming is SCIP - Solving Constraint Integer Programs

- Creat the variables

```
x = [[solver.IntVar(0,1,'x[{i}][{j}]') for j in
range(K)] for i in range(N)]
```

  As in this problem, we define an array x[i][j] : size=NxK, whose value is 1 if customer[i] is served by truck[j]

- Define the constraints

  The first constraint is each customer must be served by exactly 1 truck.

```
for i in range(N):
    solver.Add(sum(x[i]) <= 1)
```

  The next constraint is the amount packed in each bin is greater than lower bound and less than upper bound.

```
for j in range(K):
    solver.Add(sum[x[i][j]*D[i] for i in
range(N)] >= c1[j])
    solver.Add(sum[x[i][j]*D[i] for i in
range(N)] <= c2[j])
```

- Define the objective

```
obj = []
for j in range(K):
    for i in range(N):
        obj.append(x[i][j]*c[i])
solver.Maximize(sum(obj))
```

  The goal is maximizing the total benefit, which is calculated by the total sum of each item value.

- Call the solver and print the solution

4. **Ortools CP-SAT (Tuan)**
   CP-SAT is another solver for linear integer programming which implements local search and meta-heuristics on top of a Constraint Programming solver. Its performance is much higher than SCIP which was used above. Moreover, we add an option to feed the solver with a initial solution from greedy algorithm, which improves the time required to find some first feasible solutions. This option is especially helpful when the time limit is low and we just need a good feasible solution. However, calling the Greedy solver and

adding initial solution might takes some time and potentially add a few seconds to the total solve time if the optimal solution was found within time limit.
To verify this property, we ran the solver a few time with N = 10000 and K = 50 with a variable time limit on Google Colab:

Time limit	Without initial solution		With initial solution	
	Value rate (%)	Deliver rate (%)	Value rate (%)	Deliver rate (%)
30	TIMED OUT	TIMED OUT	98.83	98.46
90	91.09	90.70	98.86	98.82
120	96.82	95.59	99.06	98.98

IV. Experiments

Apart from aforementioned testings, we ran 2 tests with N = 10, 50, 100; K = 2, 5 and N = 100, 1000; K = 10, 50, 100, 500, each ran 100 times with the time limit of 60 seconds. The result is shown below.
Link

N = 10, K = 2					N = 10, K = 5				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)		Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	74.28	71.91	6.0	0.0002	Greedy	85.29	84.46	35.0	0.004
Hill Climbing	81.01	77.90	0.00	60.000	Hill Climbing	93.35	91.90	0.00	60.000
Simulated Annealing	81.01	78.00	0.00	60.000	Simulated Annealing	93.23	91.70	0.00	60.000
CP SAT	81.01	78.00	0.00	0.006	CP SAT	93.49	92.00	0.00	0.021
ILP (SCIP)	81.01	77.90	0.00	0.009	ILP (SCIP)	93.49	92.00	0.00	0.008
CP SAT + Greedy	81.01	77.90	0.00	0.005	CP SAT + Greedy	93.49	92.00	0.00	0.028
N = 50, K = 2					N = 50, K = 5				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)		Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	76.99	71.24	0.00	0.0004	Greedy	85..79	83.12	0.00	0.005
Hill Climbing	77.89	74.80	0.00	60.000	Hill Climbing	91.43	88.36	0.00	60.000
Simulated Annealing	82.60	78.62	0.00	60.000	Simulated Annealing	92.12	89.06	0.00	60.000
CP SAT	82.92	79.32	0.00	1.450	CP SAT	93.44	91.22	0.00	22.445
ILP (SCIP)	82.92	79.26	0.00	0.039	ILP (SCIP)	93.47	91.22	0.00	1.383
CP SAT + Greedy	82.92	79.34	0.00	1.360	CP SAT + Greedy	93.47	91.22	0.00	22.396
N = 100, K = 2					N = 100, K = 5				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)		Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	78.41	71.86	0.00	0.007	Greedy	89.69	85.74	0.00	0.007
Hill Climbing	79.07	76.15	0.00	60.000	Hill Climbing	92.50	90.11	0.00	60.000
Simulated Annealing	81.64	77.87	0.00	60.000	Simulated Annealing	92.37	90.14	0.00	60.000
CP SAT	83.11	79.25	0.00	0.171	CP SAT	93.20	93.20	0.00	11.606
ILP (SCIP)	83.11	79.27	0.00	0.098	ILP (SCIP)	93.20	93.20	0.00	1.940
CP SAT + Greedy	83.11	79.27	0.00	0.374	CP SAT + Greedy	93.20	93.20	0.00	11.973

N = 100, K = 10				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	92.14	90.68	1.00	0.009
CP SAT	97.20	96.03	0.00	2.183
ILP (SCIP)	97.20	96.03	0.00	6.492
CP SAT + Greedy	97.20	96.03	0.00	2.379

N = 100, K = 50				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	100.00	100.00	98.00	0.006
CP SAT	100.00	100.00	15.00	12.801
ILP (SCIP)	100.00	100.00	39.00	32.892
CP SAT + Greedy	100.00	100.00	13.00	11.895

N = 100, K = 100				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	0.00	0.00	100.00	0.003
CP SAT	100.00	100.00	61.00	25.848
ILP (SCIP)	100.00	100.00	74.00	15.450
CP SAT + Greedy	100.00	100.00	61.00	26.021

N = 100, K = 500				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	0.00	0.00	100.00	0.012
CP SAT	100.00	100.00	10.00	24.180
ILP (SCIP)	100.00	100.00	26.00	22.583
CP SAT + Greedy	100.00	100.00	10.00	23.803

N = 1000, K = 10				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	94.39	91.59	0.00	0.018
CP SAT	96.37	94.82	0.00	19.061
ILP (SCIP)	96.17	94.69	0.00	39.874
CP SAT + Greedy	96.36	94.82	0.00	27.999

N = 1000, K = 50				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	99.32	99.02	0.00	0.013
CP SAT	99.75	99.62	0.00	28.603
ILP (SCIP)	99.32	99.33	16.22	59.861
CP SAT + Greedy	99.75	99.62	0.00	24.803

N = 1000, K = 100				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	99.88	99.82	95.00	0.029
CP SAT	99.94	99.94	42.00	30.433
ILP (SCIP)	TIMED OUT	TIMED OUT	TIMED OUT	TIMED OUT
CP SAT + Greedy	99.93	99.93	41.00	29.586

N = 1000, K = 500				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	0.00	0.00	100	0.177
CP SAT	TIMED OUT	TIMED OUT	TIMED OUT	TIMED OUT
ILP (SCIP)	TIMED OUT	TIMED OUT	TIMED OUT	TIMED OUT
CP SAT + Greedy	TIMED OUT	TIMED OUT	TIMED OUT	TIMED OUT

N = 10000, K = 10				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	94.94	90.88	0.00	0.068
CP SAT	96.12	94.60	0.00	TIMED OUT
ILP (SCIP)	TIMED OUT	TIMED OUT	TIMED OUT	TIMED OUT
CP SAT + Greedy	96.22	94.65	0.00	TIMED OUT

N = 10000, K = 50				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	98.85	98.56	0.00	0.076
CP SAT	TIMED OUT	TIMED OUT	TIMED OUT	TIMED OUT
ILP (SCIP)	TIMED OUT	TIMED OUT	TIMED OUT	TIMED OUT
CP SAT + Greedy	98.85	98.56	0.00	TIMED OUT

N = 10000, K = 100				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	99.59	99.39	0.00	0.108
CP SAT	TIMED OUT	TIMED OUT	TIMED OUT	TIMED OUT
ILP (SCIP)	TIMED OUT	TIMED OUT	TIMED OUT	TIMED OUT
CP SAT + Greedy	99.59	99.39	0.00	TIMED OUT

N = 10000, K = 500				
	Value rate (%)	Deliver rate (%)	Failing rate (%)	Exec time (s)
Greedy	0.00	0.00	100	0.642
CP SAT	TIMED OUT	TIMED OUT	TIMED OUT	TIMED OUT
ILP (SCIP)	TIMED OUT	TIMED OUT	TIMED OUT	TIMED OUT
CP SAT + Greedy	TIMED OUT	TIMED OUT	TIMED OUT	TIMED OUT

V. Conclusion

Overall, all proposed methods are able to solve the problem and even give the optimal solutions. However, the performance and time taken to solve differ between them.

The greedy algorithm gives feasible solutions in the shortest time and its solutions are quite impressive with a high rate of total values delivered, which also increases as N increases. However, its failure rate is high when N and K are close to each other.

Simulated Annealing generally performs slightly better than Hill Climbing, but the performance gap gets narrower when K increases. It may take a while for these methods to give feasible solutions but when provided with enough time, they can return the optimal ones. Furthermore, these algorithms are implemented in Python which is famous for its poor speed. Their performance could be better if implemented in other programming languages.

ILP method using OR-TOOLS (SCIP solver) returns the optimal solution in the shortest time when N and K are small enough. But when N and K are higher (N >100, K> 10), CP using OR-TOOLS (CP-SAT solver) solves the problem in a shorter time, especially when we only need feasible solutions. However, when N and K are enormous, it might take a long time. In this case, CP with an initial solution might be a better bet as the performance gets much better if Greedy can provide us with a feasible solution.

In conclusion, methods using OR-Tools give better performance given the same time limit compared to others. However, when N and K are large, Greedy algorithm is the recommended method to start as it potentially gives us a fairly good solution in just a few seconds - even when it fails, we didn’t waste too much time. Local search methods, even though they might give optimal or good feasible solutions, take too much time because of their programming language. They need further research to be implemented in other languages which have better performance before becoming viable.