WAREHOUSE PICKING PROBLEM

Fundamentals of Optimization

Group 4

APPROACHES

What do we want to present in this slides?

Problem description

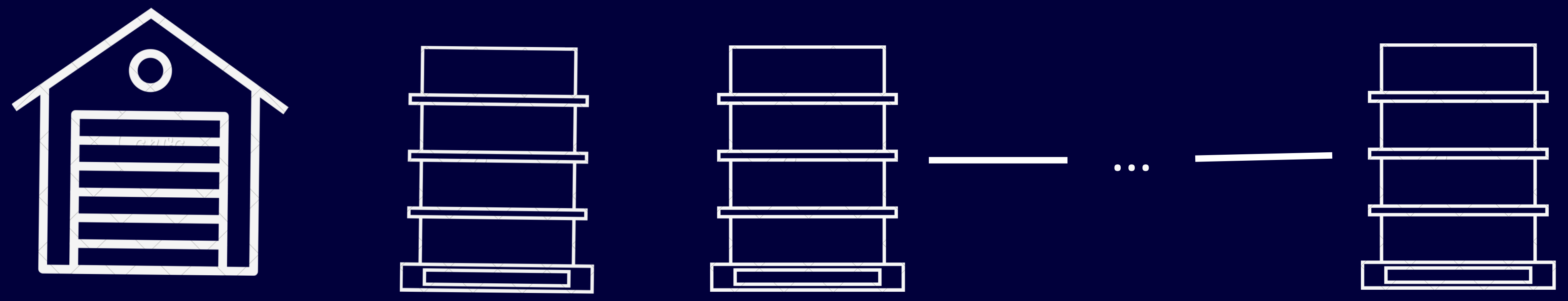
Modelling

Algorithms

Experiments

Problem description

Warehouse picking problem



Input:

- M shelves (not including the depot)
- N types of products
- Matrix D, Q, q

Output: p_min - the shortest path to take enough products for our order

	0	1	...	M
0	0	3	...	2
1	3	0	...	7
...	...	...	...	...
M	2	7	...	0

D[(M+1)x(M+1)]: Distance matrix

d(i, j) = distance from shelf i to shelf j

	0	1	...	M-1
0	4	7	...	5
1	12	9	...	7
...	...	...	...	...
N-1	4	6	...	3

Q[NxM]: Quantity matrix

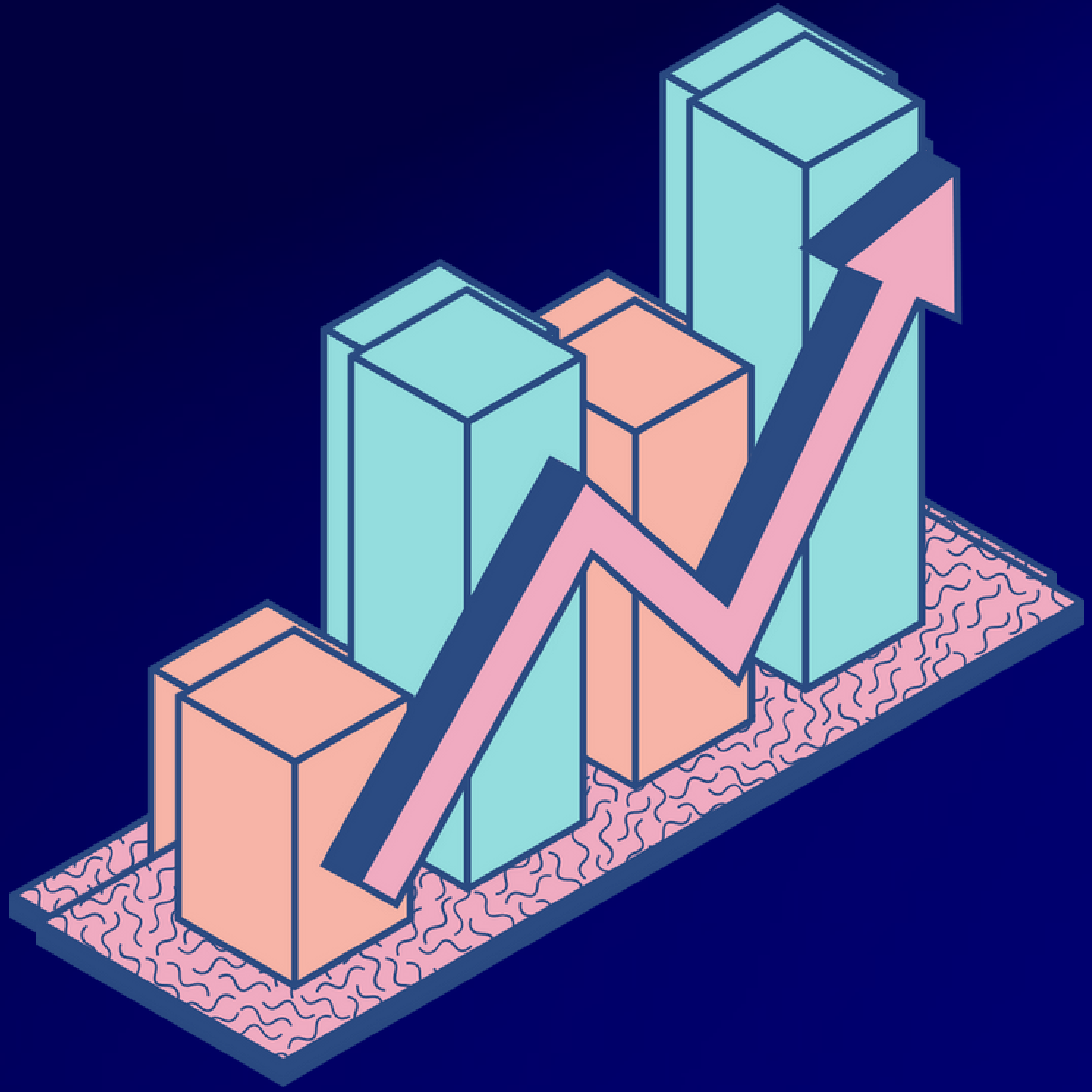
Q(i, j) = The number of products of type i displayed on shelf j

0	1	...	N-1
6	3	...	2

q[Nx1]: Order matrix

q(i) = the number of products of type i need to take for the order

Modeling



Initialize variable

x_{ij} : decision variable indicating whether picker travel from shelf i to shelf j

p : Total distance of the routing

q_k : Number of product k that the picker need to pick up

Q_{ki} : Number of product k in the shelf i

D_{ij} : The distance between shelf i to shelf j

Modeling

1. The picker must start and end at the depot.

$$\exists i, x_{0i} = 1$$

$$\exists j, x_{j0} = 1$$

2. A shelf is traveled no more than once.

$$\sum_{i=0}^{M} x_{ij} \leq 1$$

$$\sum_{j=0}^{M} x_{ij} \leq 1$$

$$\sum_{i,j=0}^{M} x_{ij} - \sum_{j,k=0}^{M} x_{jk} = 0$$

3. The total distance is the sum of all path to pick up all goods he needs.

$$p = \sum_{i,j} D_{ij} \quad \text{where } x_{ij} = 1$$

Modeling

4. When he comes to shelf j, all goods of any types he need at this shelf is picked up.

$$\text{If } \exists x_{ij} = 1, \forall k \in \{1, 2, \dots, N\}:$$

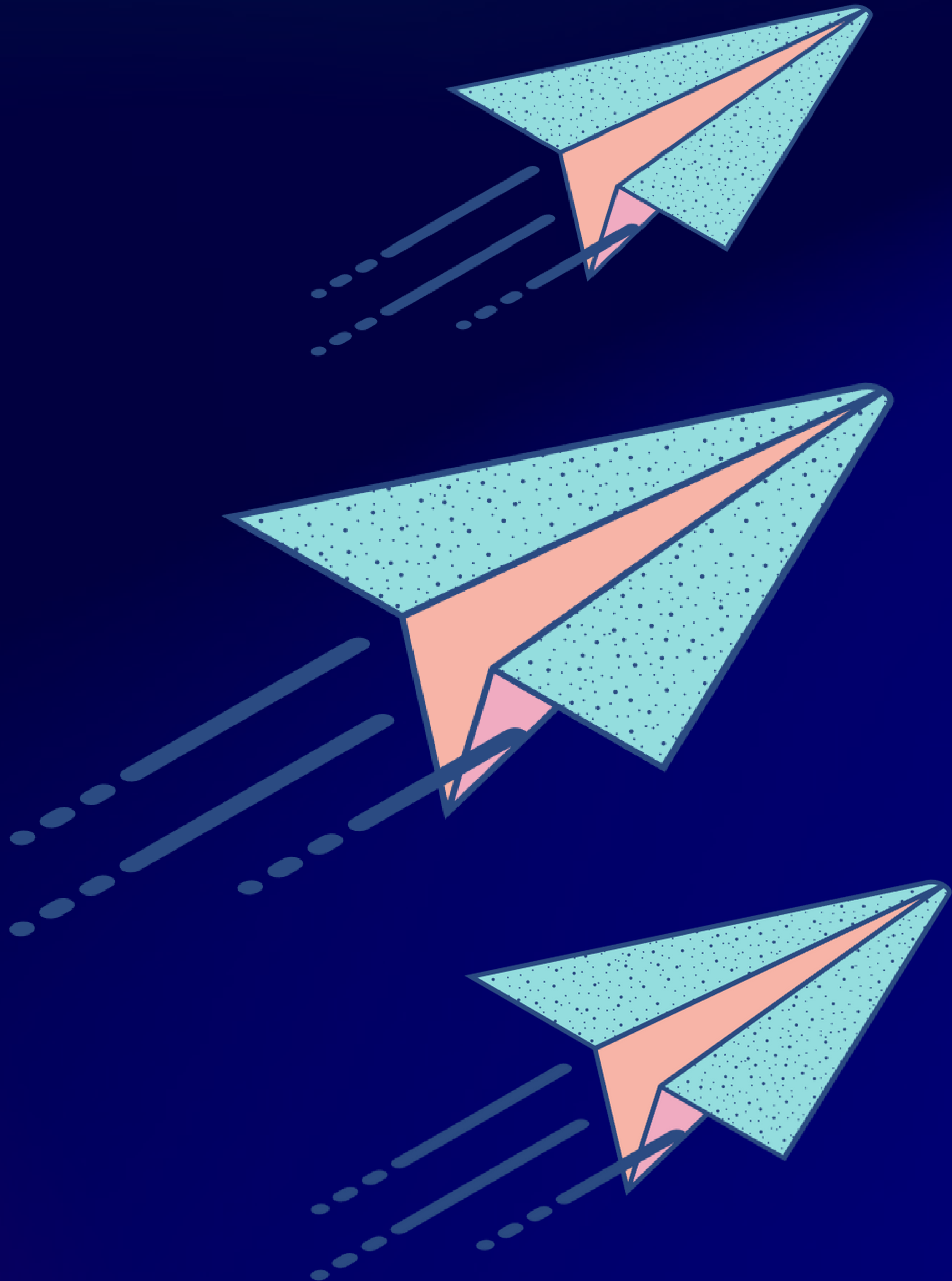
$$q_k = \begin{cases} q_k - Q_{ki}, & q_k > Q_{ki} \\ 0, & otherwise \end{cases}$$

5. When he pick up all goods he needs, come back to the depot.

$$\text{When } x_{ij}, \ \forall k \in \{1, 2, \dots, N\}: q_k = 0 \rightarrow x_{j0}$$

CSP

One of the Most optimal methods

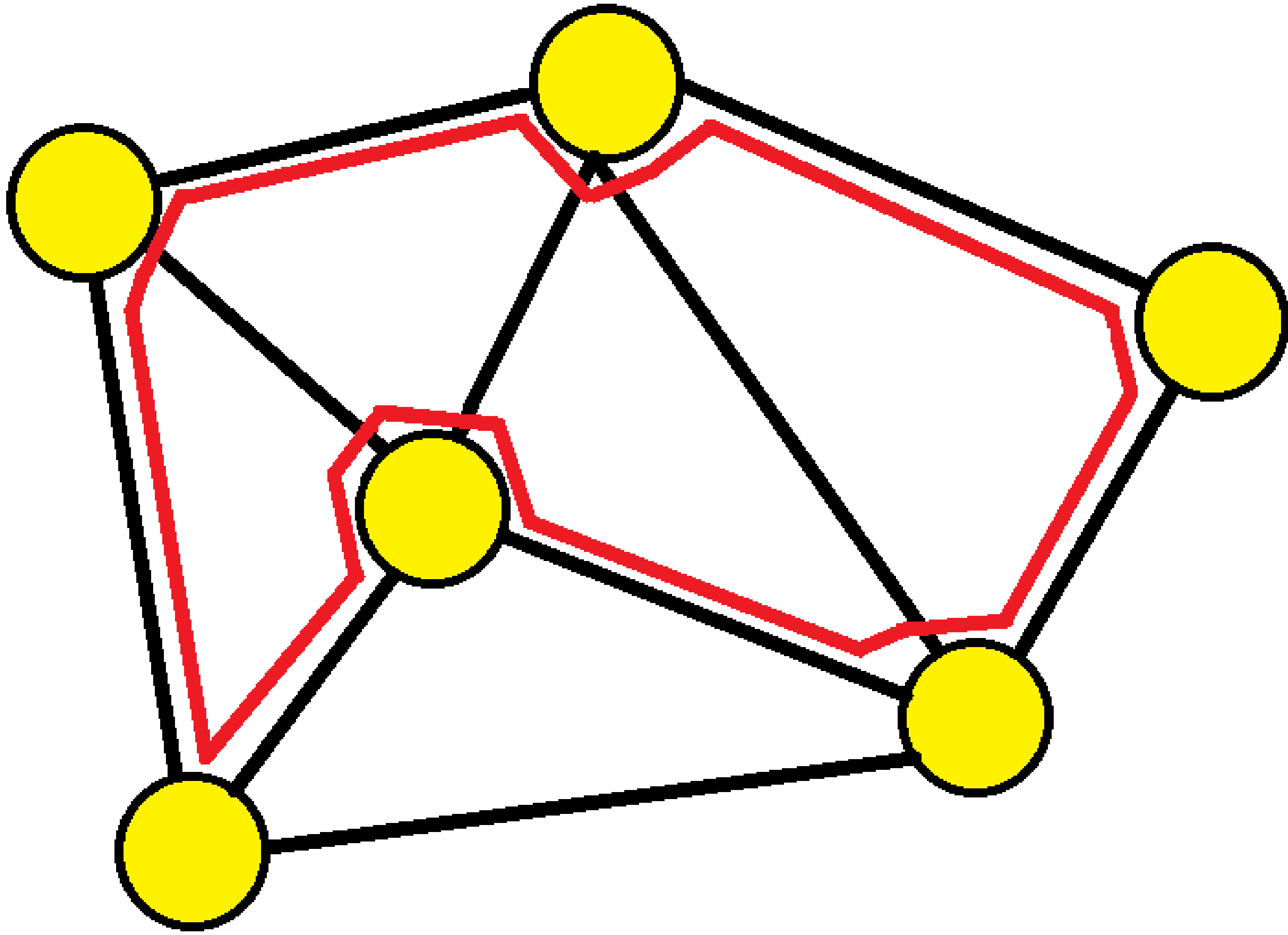


Constraints

1. The staff must start and end at the depot
2. A shelf is traveled no more than once time.

Halmintonian cycle

3. Feasible solution: The total distance is the sum of all path to pick up all goods a staff needs
4. When a staff comes to shelf m, all goods any types a picker needs at this shelf is picked up
5. When a staff picks up all needed goods, comeback to the depot
6. Optimal solution: The last result is the minimized of all feasible solution



Circuit contraint

- i : Source node
- j : Destination node
- lit : literal, TRUE if x_ij = 1

- A circuit is a unique Hamiltonian path in a subgraph of the total graph.
- A Hamiltonian path is a path in an undirected or directed graph that visits each vertex exactly once.

```
# Create the circuit constraint.
arcs = []
arc_literals = {}
for i in all_nodes:
    for j in all_nodes:
        if i == j:
            continue

        lit = model.NewBoolVar('%i follows %i' % (j, i))
        arcs.append([i, j, lit])
        arc_literals[i, j] = lit

        obj_vars.append(lit)
        obj_coeffs.append(d[i][j])
model.AddCircuit(arcs)
```

Circuit contraint meets 2 contraints of cp-sat:

1. The staff must start and end at the depot
2. A shelf is only traveled no more than once time

Distance of routing:

2 contraints about distance:

- *Feasible solution: The total distance is the sum of all journey to pick up all goods a staff needs*
- *Optimal solution: The last result is the minimized of all feasible solution (p_min)*

$$p = \sum_{i,j} D_{ij} \quad \text{where } x_{ij} = 1$$

```
#Minimize the total distance
model.Minimize(sum(obj_vars[i] * obj_coeffs[i] for i in range(len(obj_vars))))
```

Constraints for picker to pick up goods:

```
#contraint 1:
for i in range(N):
    if q[i] > Q[i][current_node-1]:
            q[i]= q[i] - Q[i][current_node-1]
    else:
            q[i] = 0
```

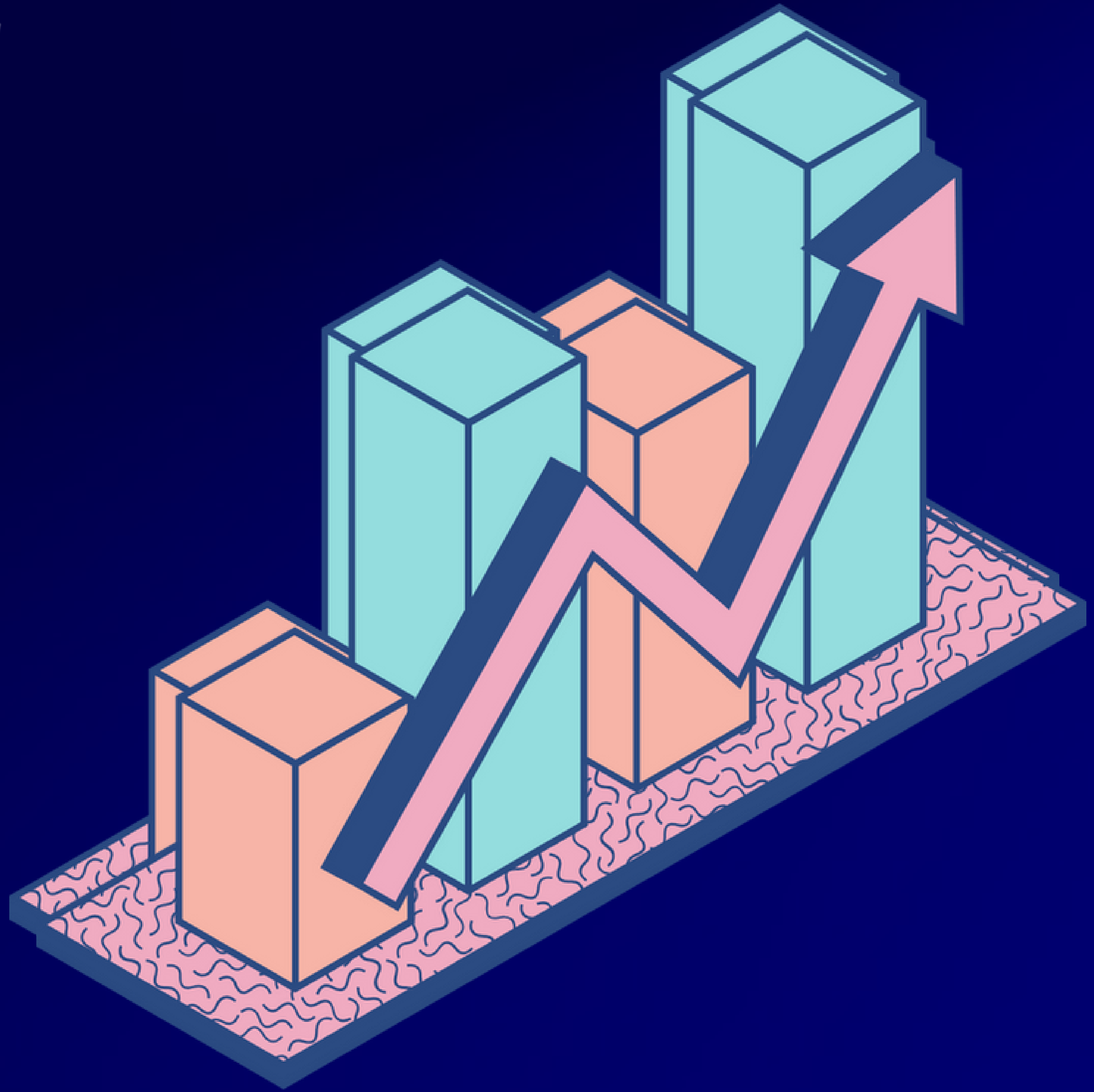
There are 2 constraints:

1. The picker will pick all needed products in the shelf that he comes.
2. Picker will come back to the depot when all needed product are picked .

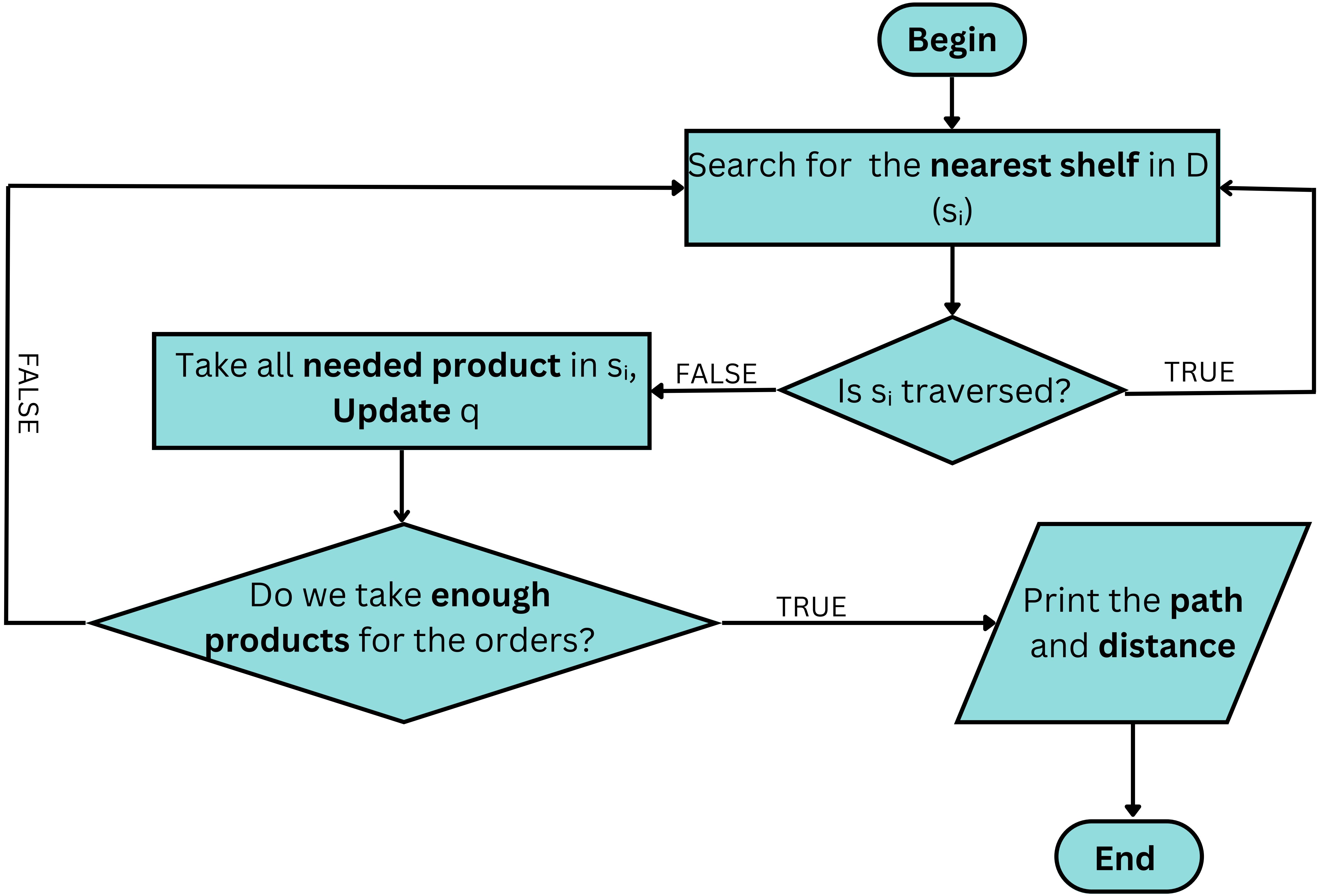
```
#contraint 2:
if all(ele==0 for ele in q):
        route_distance +=d[current_node][0]
        current_node = 0
        point.append(current_node)
        route_is_finished = True
        break

else:
        route_distance +=d[current_node][i]
        current_node  = i
        point.append(current_node)
```

Nearest Neighbor

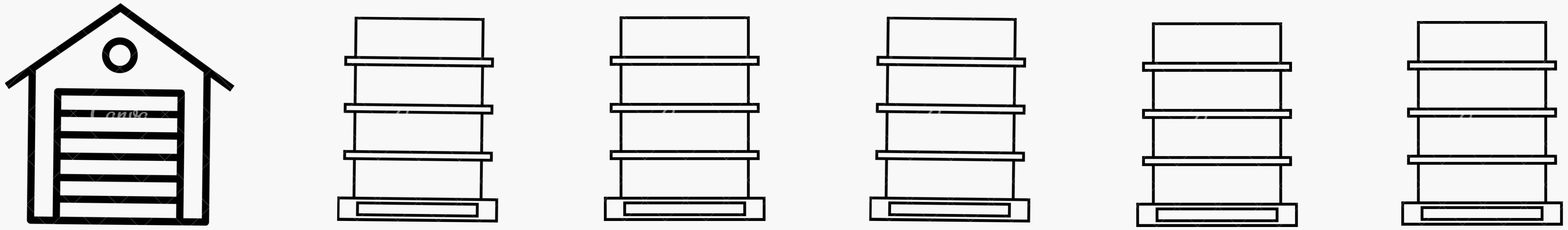
Easy and fast!



WHAT IS NEAREST NEIGHBOR ALGORITHM?



Example For the Nearest Neighbor



Shelf 0(depot) | Shelf 1 | Shelf 2 | Shelf 3 | Shelf 4 | Shelf 5

M = 5, N = 6

0	16	10	13	13	19
16	0	8	3	19	5
10	8	0	7	23	11
13	3	7	0	16	6
13	19	23	16	0	22
19	5	11	6	22	0

MATRIX D[6x6]

3	2	2	4	2
4	3	7	3	5
6	7	2	5	4
2	3	3	2	1
2	5	7	6	1
7	2	1	6	5

MATRIX Q[6x5]

8	7	4	8	11	13

MATRIX q

Example For the Nearest Neighbor

Starting from **the depot**

	0	1	2	3	4	5
	0	16	**10**	13	13	19
1	16	0	8	3	19	5
2	10	8	0	7	23	11
3	13	3	7	0	16	6
4	13	19	23	16	0	22
5	19	5	11	6	22	0

3	**2**	2	4	2
4	**3**	7	3	5
6	**7**	2	5	4
2	**3**	3	2	1
2	**5**	7	6	1
7	**2**	1	6	5

8	7	4	8	11	13

UPDATE q

6	4	0	5	6	11

Path

0	**2**

Distance 0 + 10 = **10**

Example For the Nearest Neighbor

Next, starting from the **shelf 2**

	0	1	2	3	4	5
0	0	16	10	13	13	19
1	16	0	8	3	19	5
2	10	8	0	**7**	23	11
3	13	3	7	0	16	6
4	13	19	23	16	0	22
5	19	5	11	6	22	0

3	2	**2**	4	2
4	3	**7**	3	5
6	7	**2**	5	4
2	3	**3**	2	1
2	5	**7**	6	1
7	2	**1**	6	5

6	4	0	5	6	11

UPDATE q

4	0	0	2	0	10

Path

0	2	**3**

Distance 10 + 7 = **17**

Example For the Nearest Neighbor

	0	1	2	3	4	5
0	0	16	10	13	13	19
1	16	0	8	3	19	5
2	10	8	0	7	23	11
	13	**3**	7	0	16	6
4	13	19	23	16	0	22
5	19	5	11	6	22	0

3	2	2	4	2
4	3	7	3	5
6	7	2	5	4
2	3	3	2	1
2	5	7	6	1
7	2	1	6	5

4	0	0	2	0	10

UPDATE q

1	0	0	0	0	3

Path

0	2	3	**1**

Distance 17 + 3 = **20**

Example For the Nearest Neighbor

	0	1	2	3	4	5
0	0	16	10	13	13	19
	16	0	8	3	19	**5**
2	10	8	0	7	23	11
3	13	3	7	0	16	6
4	13	19	23	16	0	22
5	**19**	5	11	6	22	0

3	2	2	4	**2**
4	3	7	3	**5**
6	7	2	5	**4**
2	3	3	2	**1**
2	5	7	6	**1**
7	2	1	6	**5**

1	0	0	0	0	3

UPDATE q

0	0	0	0	0	0

Taking **enough products** for the order

Go **back** to **the depot**

Path

0	2	3	1	**5**	**0**

Distance 20 + 5 + 19 = **44**

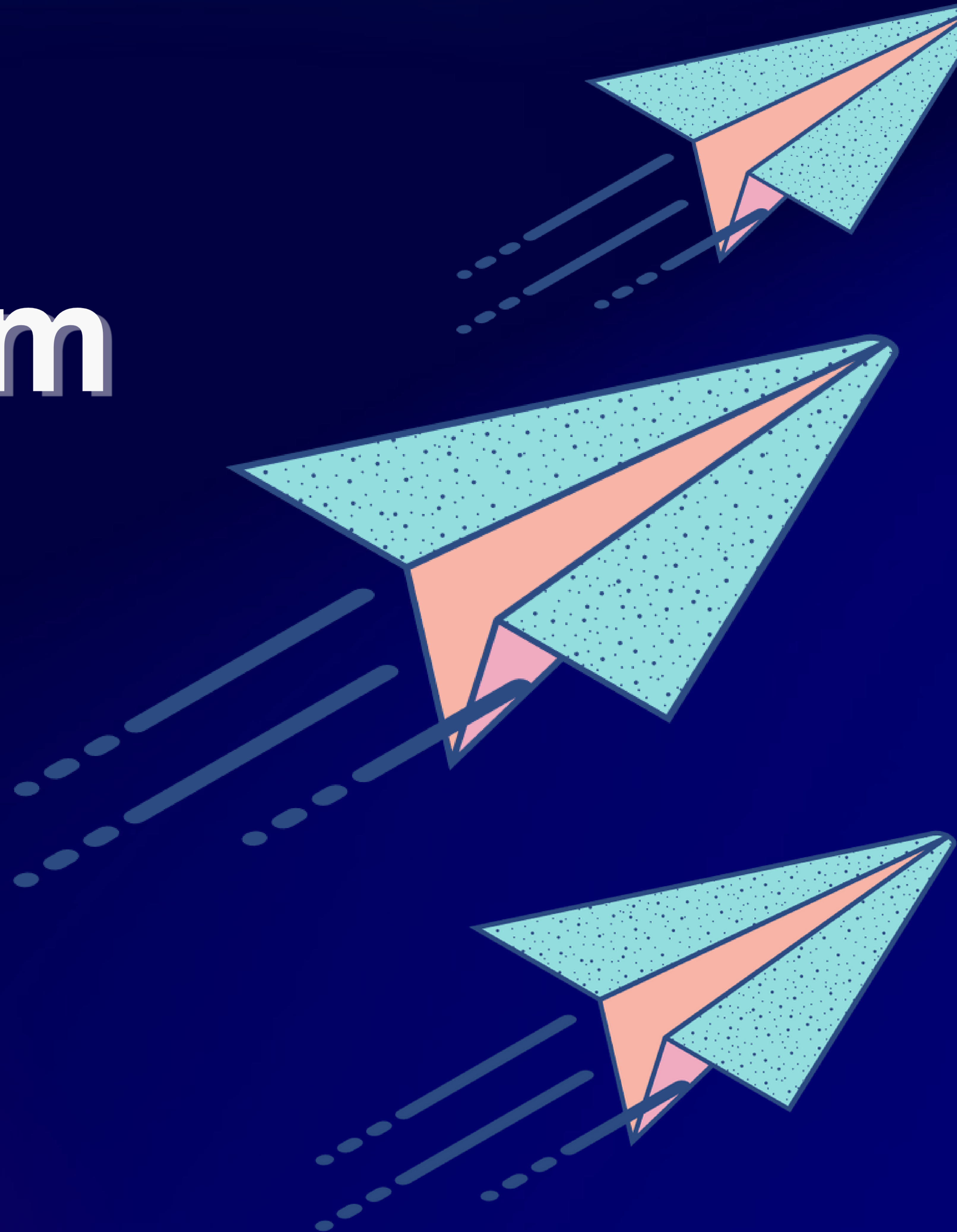
PSEUDO CODE

```
// Check if we have taken enough goods for the
order

function end(q){
   for (p_num in q){
      if (p_num != 0) return 0
      else return 1
   }
}



// Find the nearest shelf from the current location.

function cal_nearest_shelf(c, D, M, r){
   int min_d = ∞
   int nearest = -1
   for (int i = 0; i < M+1; i++){
      if (i not in r && D[c][i] != 0 && min_d > D[c][i]){
         nearest = i
         min_d = D[c][i]
      }
   }
   return nearest, D[c][nearest]
}
```

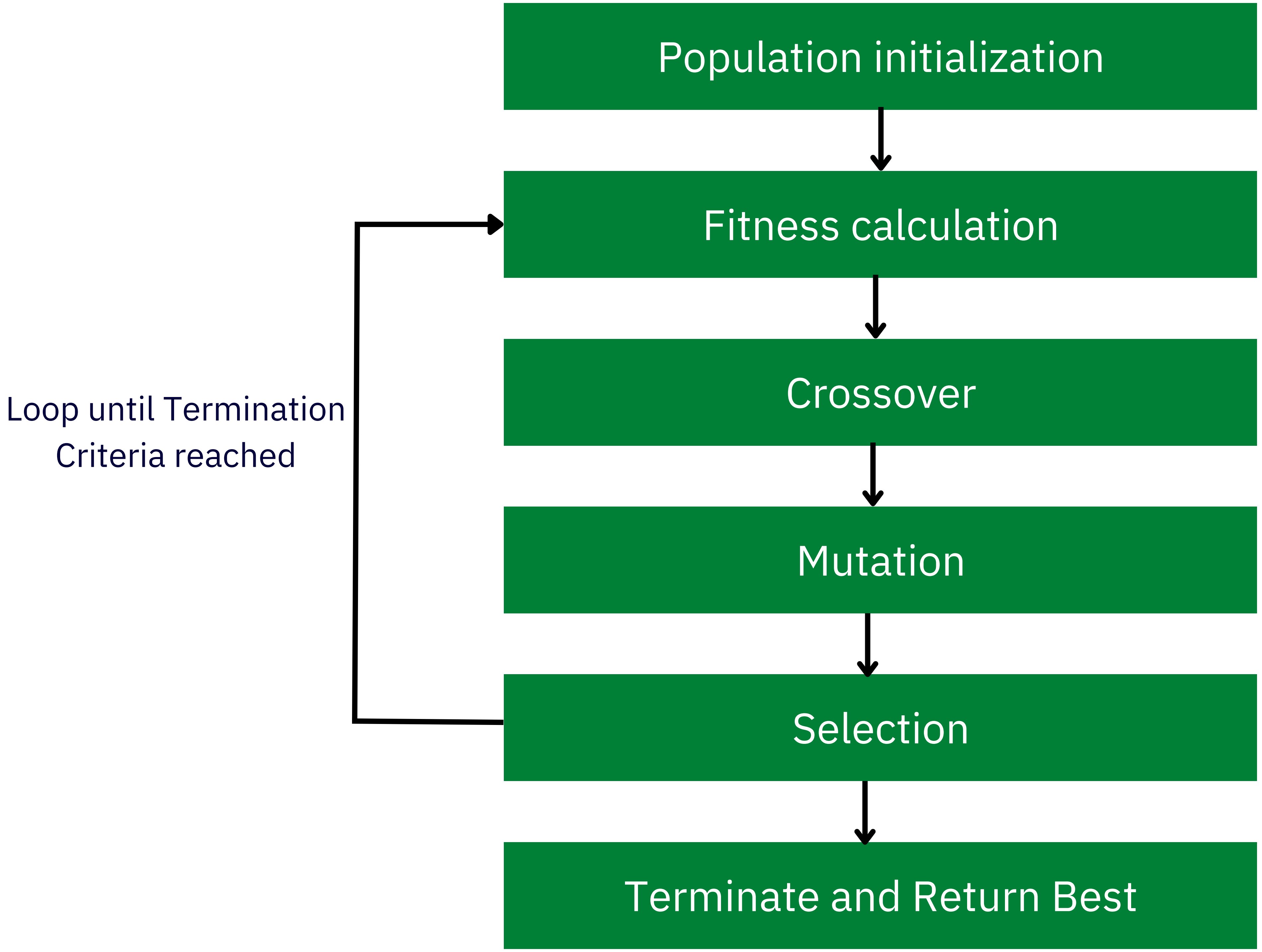
```
// Find and return the path and the distance moved

function nearest_neighbor(N, M, Q, D, q){
   int s = 0, c = s, total = 0
   r = []

   // Starting from the depot
   r.append(s)
   while (!end(q)){
      c, distance = cal_nearest_shelf(c, D, M, r)
      r.append(c)
      total += distance
      Update q // update the number of goods that need to be picked up
   }

   // After taking enough goods, return to the depot
   r.append(s)
   total += D[c][s]
   return r, total
}
```

Genetic Algorithm

Is biology your major?
You may love this algorithm!



HOW GENETIC ALGORITHM WORKS?

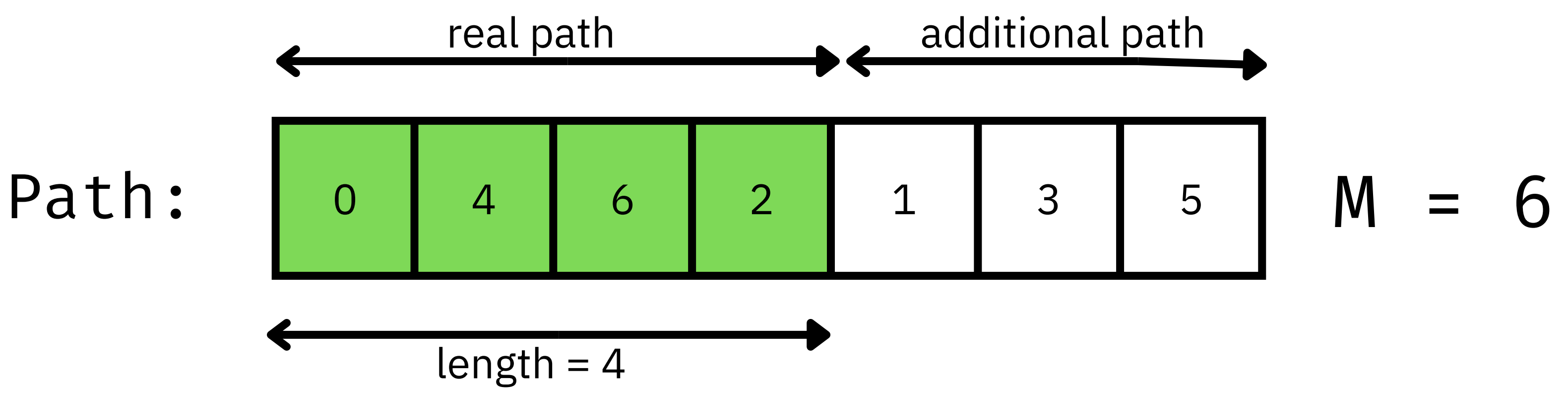


- individual: A feasible solution
- population: A set of individuals

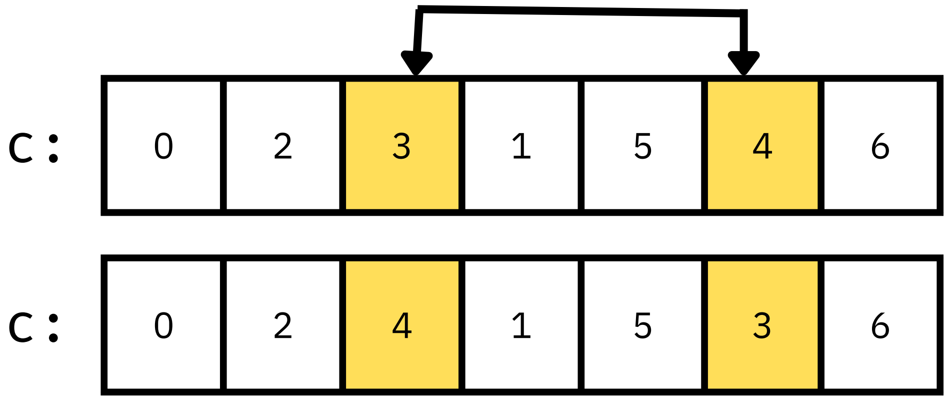
- POP_SIZE: The size of population
- gen_thres: The number of generations (termination criteria)

DATA STRUCTURE AND OPERATORS

```
// Data structure
Individual{
      path,
      fitness,
      length
}
```

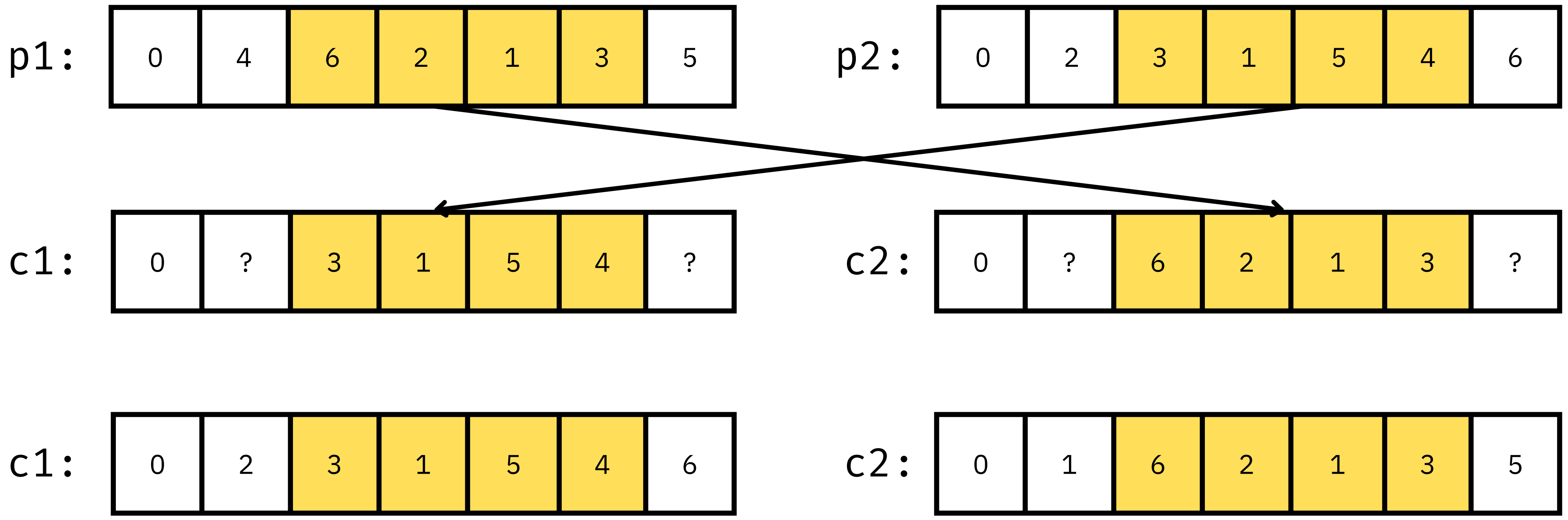


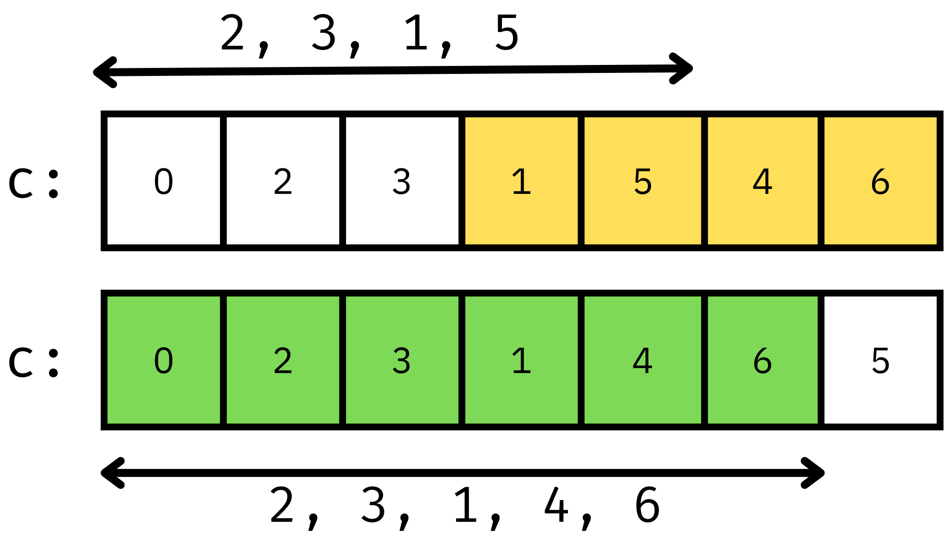

```
path = [0, 4, 6, 2, 1, 3, 5]
fitness = d(0,4) + d(4,6) + d(6,2) + d(2,0)
length = 4
```



Mutation (swap i=2 and i=5)

Crossover OX: from index = 2 -> 5





Mutation (cut from i=3)

PSEUDO CODE

```
function main(POP_SIZE, gen_thres){
     pop = []
     gen = 0
     init_pop(pop, POP_SIZE)
     while gen < gen_thres{
          shuffle(pop)
          while not end_crossover(){
               p1, p2 = select_parents()
               c1, c2 = crossover(p1, p2)
               c1 = mutation(c1)
               c2 = mutation(c2)
               pop.append(c1, c2)
          selection(pop, POP_SIZE)
          gen += 1
     print_information(pop[0])
}
```

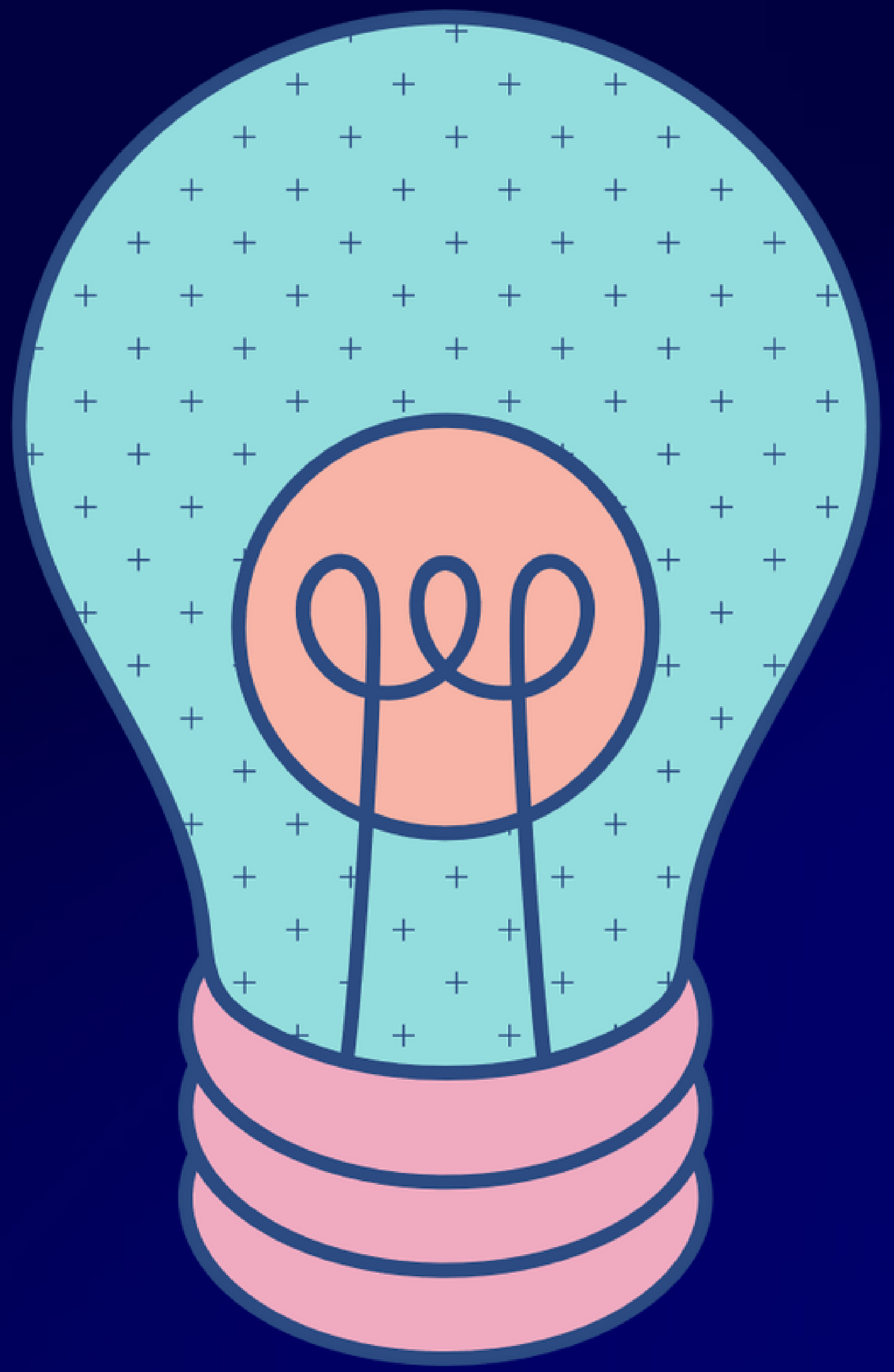
```
function mutation_swap(c){
     r = random()
     r1 = random()
     if r != r1{
          temp = c[r]
          c[r] = c[r1]
          c[r1] = temp
     }
     // Recalculate fitness and real length
     return c
}
```

```
function crossover(p1, p2){
     r = random()
     r1 = random()
     if r != r1{
          // Perform ox operators
     }
     //Calculate fitness and real length for c1, c2
     return c1, c2
}
```

```
function mutation_cut(c){
     r = random()
     path = c.path[:r]
     while not end(){
          shelf = random()
          if not is_traversed(shelf){
               path.append(shelf)
          }
      c.path = path
     //Fill the path with the remaining shelves
     // Recalculate fitness and real length
     }
     return c
}
```

Heuristics

Useful method with large-scale problems



Combining 2 ideas using local search heuristic in routing solver - ortools:

- Travelling Salesman Problem

To find the shortest route for one picker. And picker needs to visit all shelf.

- Capacity Vehicle Routing Problem

To add demand for picker to pick needed products.

Routing model

```
#routing model
manager = pywrapcp.RoutingIndexManager(len(d), num_vehicles, depot)
routing = pywrapcp.RoutingModel(manager)
```

- Index manager (manager) with 3 inputs:
  - The number of rows of the distance matrix, which is the number of locations (including the depot).
  - The number of vehicles in the problem.
  - The node corresponding to the depot.

- Routing model (routing)

Distance callback (distance_callback)

Create a function for Routing model to takes any pair of locations and returns the distance between them.

Registers Distance callback with the solver:.

```
transit_callback_index = routing.RegisterTransitCallback(distance_callback)
```

Set the cost of travel (distance)

Tell the solver how to calculate the cost of travel between any two locations (Distance)

```
# Define cost of each arc.
routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)
```

Set the default search parameters

```python
# Setting first solution heuristic.
search_parameters = pywrapcp.DefaultRoutingSearchParameters()
search_parameters.first_solution_strategy = (
    routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)
search_parameters.local_search_metaheuristic = (
    routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH)
```

PATH_CHEAPEST_ARC: Create an initial route for the solver by repeatedly adding edges with the least weight that don't lead to a previously visited node (other than the depot)

GUIDED_LOCAL_SEARCH: An option for local search metaheuristics; it uses guided local search to escape local minima.

Solve this problem with search parameters:

```python
# Solve the problem.
solution = routing.SolveWithParameters(search_parameters)
```

Add constraints for picker

There are 2 constraints:

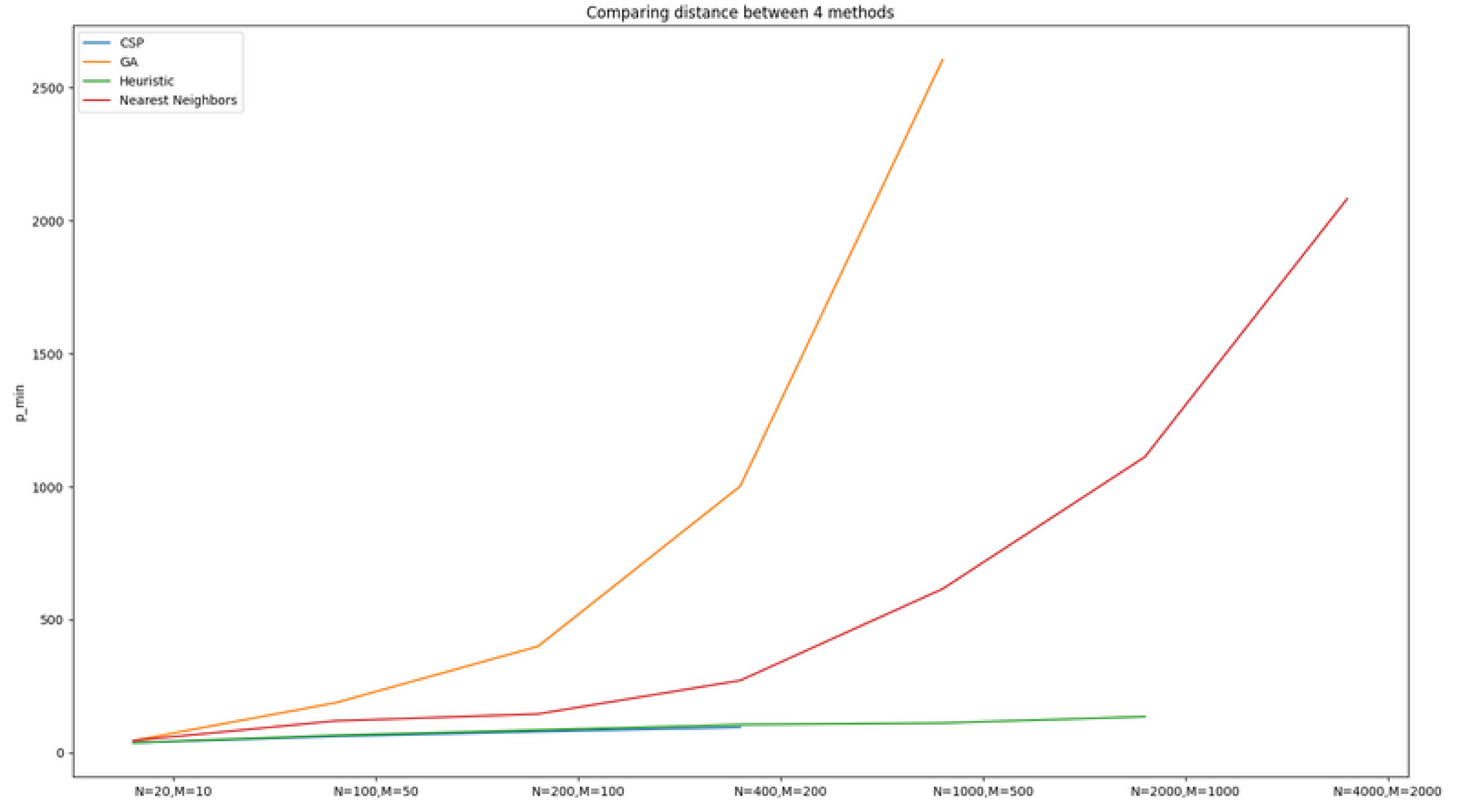
1. The picker will pick all needed products in the shelf that he comes.
2. Picker will come back to the depot when all needed product are picked .

```
#contraint 1:
for i in range(N):
    if remain_route_load[i] > Q[i][index-1]:
        remain_route_load[i]= remain_route_load[i] - Q[i][index-1]
    else:
        remain_route_load[i] = 0
```

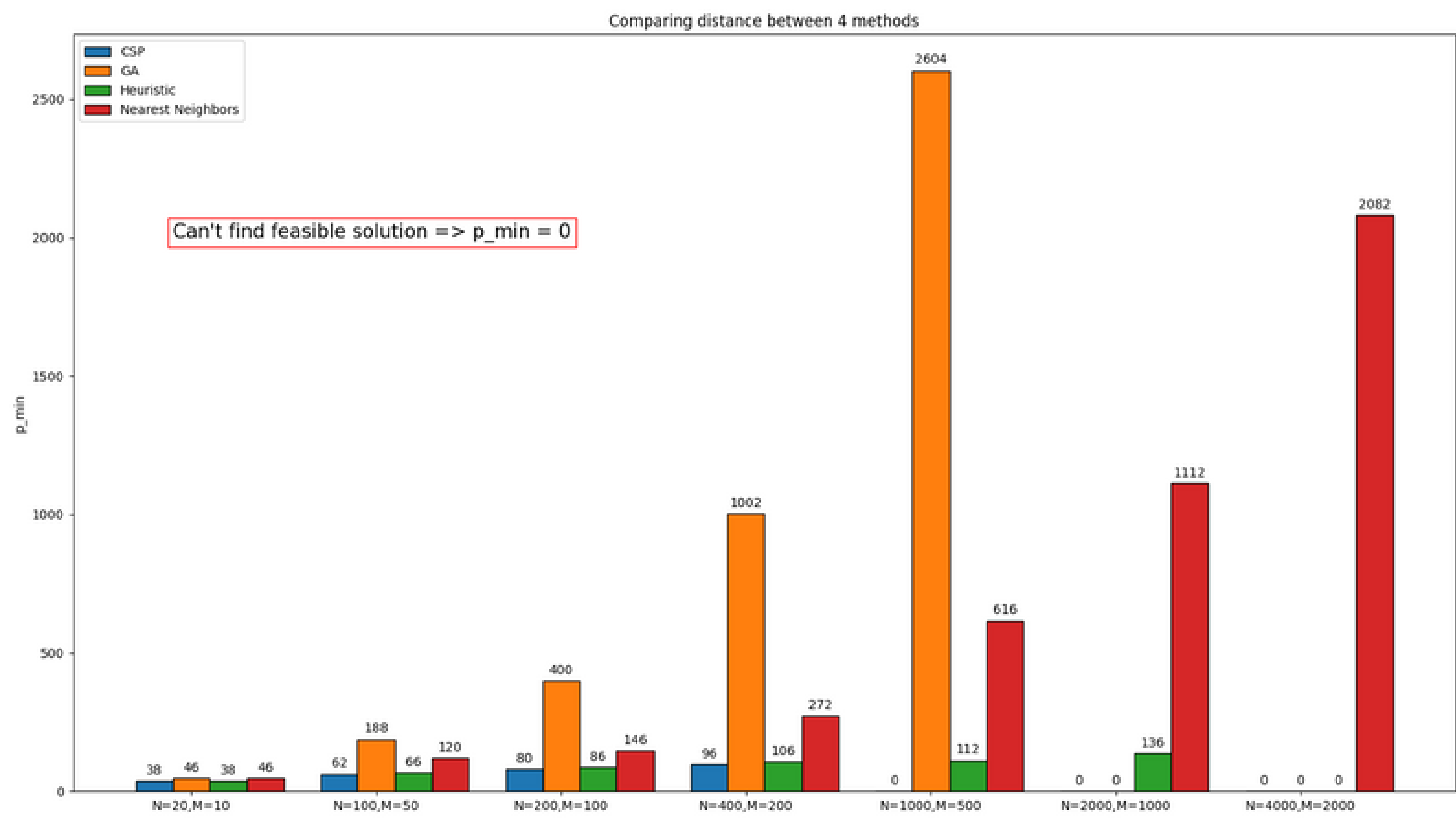
```
#constraint 2:
if all(ele==0 for ele in remain_route_load):
    previous_index = index
    index = 0
    route_distance +=d[previous_index][0]
    break
else:
    previous_index = index
    index = solution.Value(routing.NextVar(index))
    route_distance += routing.GetArcCostForVehicle(previous_index, index, 0)
```

Experiment

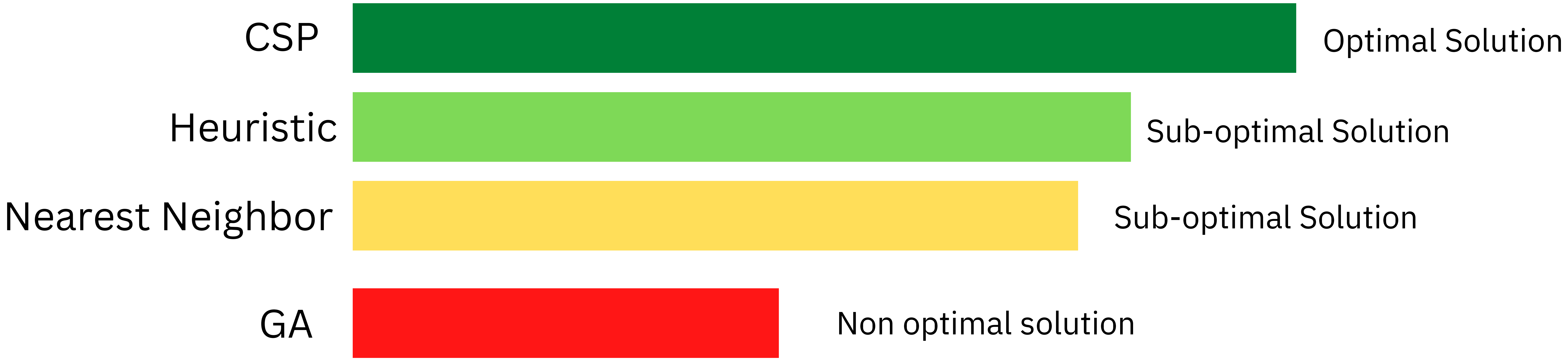
Size	CSP	GA	Heuristic	Nearest Neighbor	Optimal
N = 20 M = 10	Obj: 38 Time: 0.077739	Obj: 46 Time: 0.261499	Obj: 38 Time: 1.021601	Obj: 46 Time: 0.016001	38
N = 100 M = 50	Obj: 62 Time: 3.792506	Obj: 188 Time: 4.068183	Obj: 66 Time: 1.010069	Obj: 120 Time: 0.115547	62
N = 200 M = 100	Obj: 80 Time: 21.09461	Obj: 400 Time: 15.37137	Obj: 86 Time: 1.026791	Obj: 146 Time: 0.384347	80
N = 400 M = 200	Obj: 96 Time: 144.9656	Obj: 1002 Time: 61.61333	Obj: 106 Time: 1.098044	Obj: 272 Time: 1.196475	96
N = 1000 M = 500	Obj: N/A Time: N/A	Obj: 2604 Time: 384.3723	Obj: 112 Time: 1.715063	Obj: 616 Time: 6.712403	112
N = 2000 M = 1000	Obj: N/A Time: N/A	Obj: N/A Time: N/A	Obj: 136 Time: 4.811873	Obj: 1112 Time: 28.83697	136
N = 4000 M = 2000	Obj: N/A Time: N/A	Obj: N/A Time: N/A	Obj: N/A Time: N/A	Obj: 2802 Time: 114.9758	2802

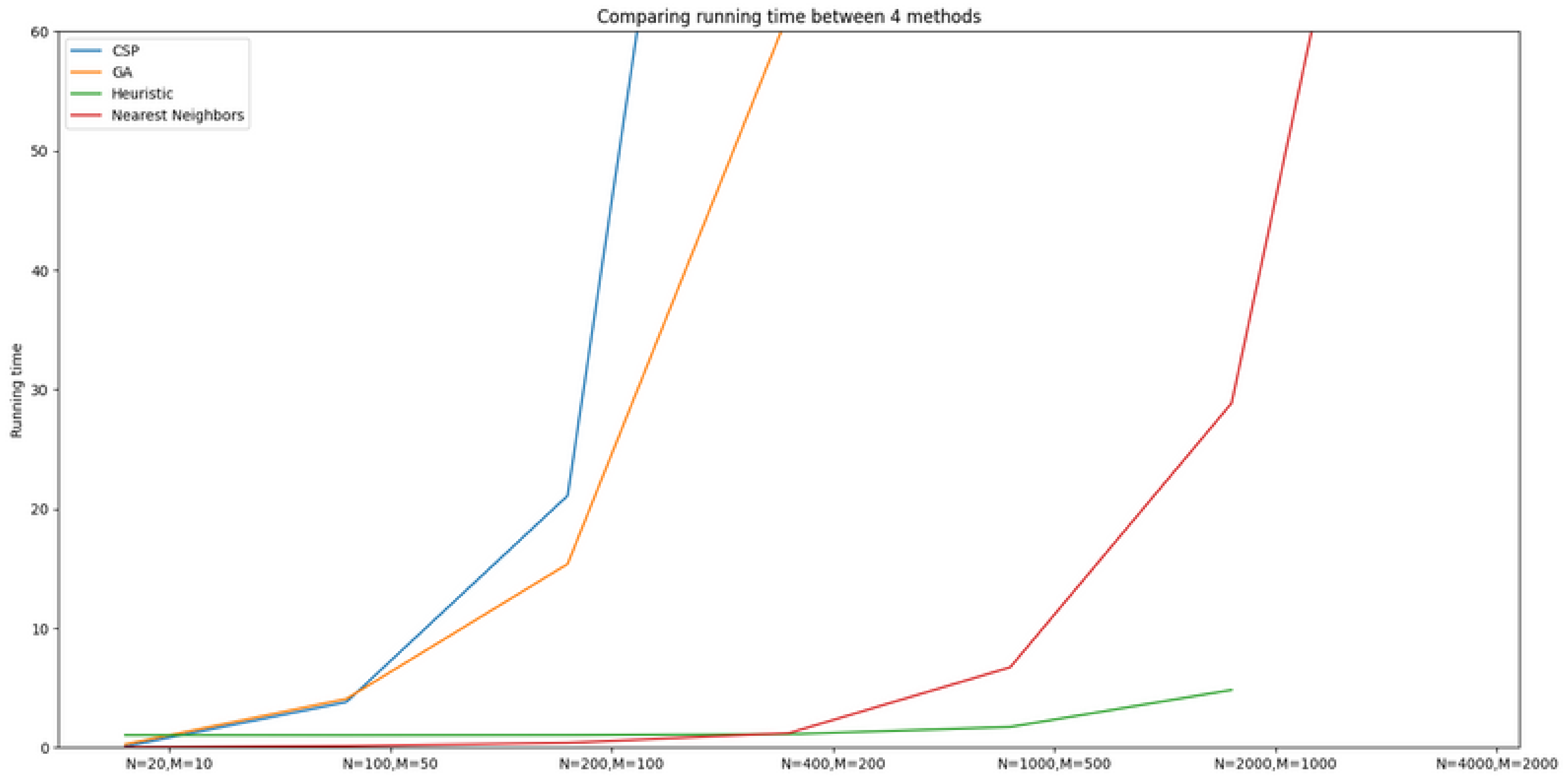


Comparing distance between 4 methods
CSP
GA
Heuristic
Nearest Neighbors
p_min
2500
2000
1500
1000
500
0
N=20,M=10
N=100,M=50
N=200,M=100
N=400,M=200
N=1000,M=500
N=2000,M=1000
N=4000,M=2000


Comparing distance between 4 methods
CSP
GA
Heuristic
Nearest Neighbors
Can't find feasible solution => p_min = 0
p_min
0
500
1000
1500
2000
2500
38
46
38
46
62
188
66
120
80
400
86
146
96
1002
106
272
0
2604
112
616
0
0
136
1112
0
0
0
2082
N=20,M=10
N=100,M=50
N=200,M=100
N=400,M=200
N=1000,M=500
N=2000,M=1000
N=4000,M=2000

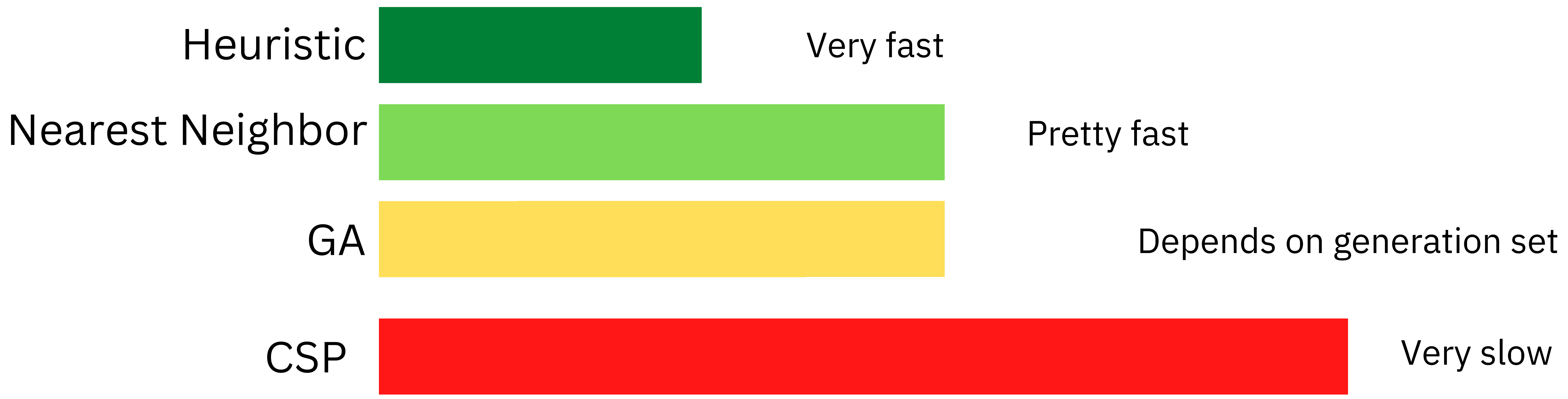
/solution /


CSP
Optimal Solution
Heuristic
Sub-optimal Solution
Nearest Neighbor
Sub-optimal Solution
GA
Non optimal solution

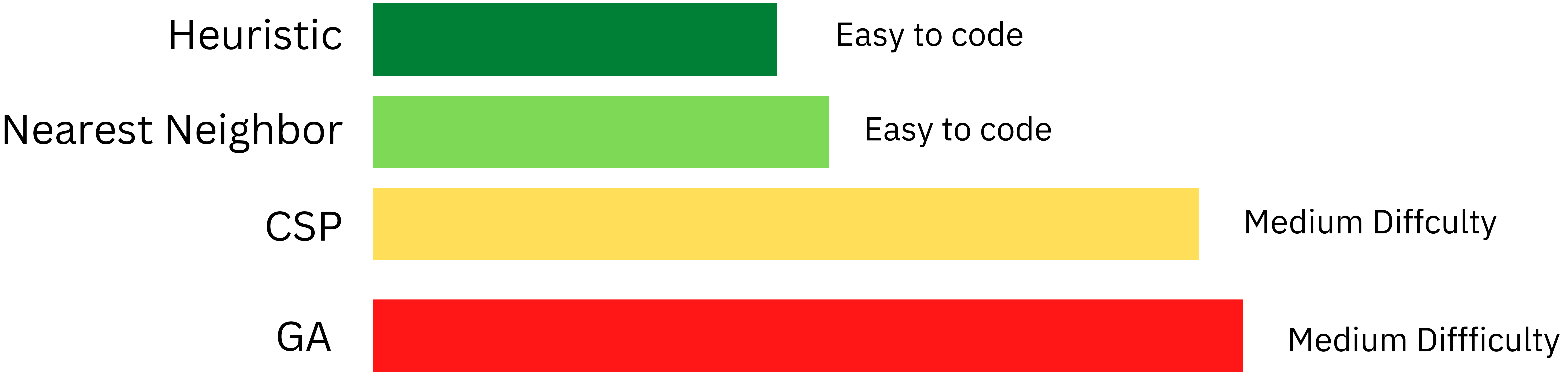


Set time limit range up to 60s

/run time/


Heuristic
Very fast
Nearest Neighbor
Pretty fast
GA
Depends on generation set
CSP
Very slow

/difficulty /


Heuristic
Easy to code
Nearest Neighbor
Easy to code
CSP
Medium Diffculty
GA
Medium Diffficulty

Group 4

THANKS!