Part I
Overview

1 Our Problem

Our given problem is Order Picking Up Route in Warehouse Problem. It is stated as followed:

- In a warehouse there are:
  - M shelves 1,2,...,M.
  - N product 1,2,..,N. The amount of product i in the shelf j is Q[i][j]
  - The distance between shelf i and j is d(i,j) ($0 \leq$ i, j $\leq$ M)
  - A warehouse staff needs to pick up enough product for customers. Total needed amount of product i is q[i] for i = 1,2,...,N
- The staff start from the door(point 0) of the warehouse, and want to visit some shelves to pick up enough products.
- Goal: Find the sequence of shelves to visit such that the total travel distance is minimal

It is clear that this is a routing problem. The problem reminds us of the Traveling Saleman Problem (TSP). In both problems, we have a lists of cities (which are basically shelves in our task) and we need to find a shortest path. However, there is a major difference. While it is necessary to visit all cities and come back to the start point in TSP, in this problem, we stop visiting and come back to point 0 as long as we pick up enough product. This has drastically changed our approach in many algorithms to solve this tasks.

2 Our Work

To tackle the problem, our group has designed and evaluated a total of 6 algorithms. Three are exact and three are heuristic. The exact algorithms includes Branch and Bound, Constraint Programming (CP) and Integer Programming (IP). The heuristic algorithms involves Greedy, Local Search combined with Simulated Annealing (LS-SA) and Ant Colony Optimization (ACO). In the next chapter in this report, the idea of each algorithm is proposed and then, in the chapter III , we analyze and compare the performance of all algorithms.

Part II
Algorithms Design

1 Branch and Bound

1.1 Method

Our Branch and Bound algorithm for this problem has some similarities with the Branch and Bound algorithm for the TSP problem. In the Branch and Bound method, we compute a bound on the best possible solution that we can get if we down this node for a current node in the state space tree. If the bound on the best possible solution is worse than the current best (best computed by far), we ignore the subtree rooted with the node.

For our problem, we consider the tree's root node to be the door - shelf 0, with the weight of edge (i, j) equal to the distance between shelves i and j. To begin, we compute the initial lower bound for the root node at level 0, and then for each subsequent level, we compute a lower bound for each node based on the current partial tour and distances to the remaining unvisited cities. The node with the smallest lower bound is then selected. The branching and bounding process is repeated until the entire solution space has been explored.

Pseudo code for our Branch and bound algorithm:

```
N = length(distance_matrix)
if we have picked up enough products then
    calculate the cost of the route
    if cost of the current path ≤ minimum_cost then
        Update minimum_cost, minimum_path
        for shelf in range (N) do
            if shelf is not visited then
                store the current kpi
                update the cost of current path
                update the bound
                if bound ≤ minimum_cost then
                    append shelf to the current path and marked as visited
                    update the remaining kpi
                    call the function again for the next level
                end if
                reset all the changes
            end if
        end for
    end if
end if
return minimum_path, minimum_cost
```

1.2 Lower Bound Function

Here is how the bound formula is defined:

Cost of the current path + (minimum cost in the distance matrix * unvisited shelves)

Since we might not have to visit all the shelves to pick up enough products, we first use the above bound formula to find a feasible solution, and then we use the following one:

Cost of the current path + (minimum cost in the distance matrix * **N**)

For **N** is the length of the current best solution minus the number of shelves that have been visited.

2 Constraint Programming and Integer Programming

2.1 Method

Here, we model the problem and use ortool to solve it. We use CPModel for CP and Pywralp for IP. Overall, we use the same model for both constraint and integer programming, the only difference is the syntax of each OR tool.

2.2 Problem Modelling

Convention: the door is shelf 0
$M = \{0, 1, 2, ..., m\}$: set of $m + 1$ shelves
$N = \{1, 2, ..., n\}$: set of n products
q_i: the amount of product i to be picked, $i \in N$
Q_{ij}: the amount of product i in shelf j $\forall i \in N, j \in M \setminus \{0\}$
d_{ij}: distance between shelf i and shelf j $\forall (i, j) \in M^2$

2.2.1 Decision variables

- $x_{ps} \in \{0, 1, ..., Q_{ps}\}$: number of product p picked in shelf s $\forall p \in N, s \in M \setminus \{0\}$

- y_{ij}: edge (i,j) is visited or not $\forall (i, j) \in M^2$,
  $y_{ij} \in \{0, 1\}$ if $i \neq j$
  $y_{ij} \in \{0\}$ if $i = j$

- $u_i \in \{2, 3, ..., m + 1\}$: Miller-Tucker variable $\forall i \in M \setminus \{0\}$

2.2.2 Objective to be minimize

$$\sum_{(i,j) \in M^2} d_{ij} \times y_{ij}$$

2.2.3 Constraints

For simplicity, denote:

$t(s) = \sum_{i=0}^{n} y_{is}$: number of edges going to s $\forall s \in M$

$f(s) = \sum_{i=0}^{n} y_{si}$: number of edges going from s $\forall s \in M$

- Total pickup is larger or equal to kpi:

$$\sum_{s=1}^{m} x_{ps} \geq q_{ps} \, \forall p \in N$$

- If a shelf is visited, we pick up all the amount of products on that shelf. Otherwise, no pickup allowed

$$x_{ps} = Q_{ps} \times f(s) \, \forall p \in N, s \in M \setminus \{0\}$$

- Ensure the number of edges going to a shelf must be equal to the numbers of edge going out from that shelf and not larger than 1.

$$f(0) = 1$$

$$t(0) = 1$$

$\forall s \in M \setminus \{0\}$

$$f(s) \leq 1$$

$$t(s) \leq 1$$

$$f(s) = t(s)$$

-Miller-Tucker-Zemlin subtour elimination:

$$u_i - u_j + 1 \leq m \times (1 - y_{ij}) \, \forall i \in M \setminus \{0\}, j \in M \setminus \{0\}$$

3 Greedy

The greedy method is a heuristic algorithmic paradigm that adheres to the problem-solving heuristic of making the locally optimal choice at each stage in the hope of discovering a global optimum. In other words, at each step, the algorithm makes the best decision it can based on the available information, without regard for the long-term consequences.

In this problem, at each step, we choose the locally optimal choice of the nearest shelf and update the solution based on this choice, with the goal of reaching a global optimum where the number of required products is satisfied.

Pseudo code for our Greedy algorithm:

$visited[shelf] = False \, \forall \, shelf$ in $shelves$
$route = []$
$value = 0$

$current\_shelf = 0$ and marked as visited
kpi = initial_kpi
while True **do**
    select the nearest shelf based on $current\_shelf$
    update $value$
    update $current\_shelf$ and marked as visited
    append $current\_shelf$ to $route$
    update the remaining kpi
    if $all(kpi == 0)$ **then**
        update $value$
        break the loop
    end if
end while
return $route$, $value$

4 Ant Colony Optimization

4.1 Introduction

Ant Colony is a metaheuristic algorithm inspired by foraging behaviour of ants in nature. The underlying principle of ACO is to observe the movement of the ants from their nests in order to search for food in the shortest possible path. Initially, ants move randomly in search of food around their nests and leave pheromone on their path. Pheromone is a substance generated by ants for communication between them. Ants will choose path based on the concentration of pheromone in each path. Ants generate more pheromone for better path and the path with more pheromone will be likely to followed by next ants. Learning from that interesting natural feature of ants, ACO can be used to solve many routing problem including our problem.

4.2 Method

Our ACO algorithm for this problem shares many similarities to ACO for TSP. Firstly, we have a number of ants to visit the shelves. An ant acts like the warehouse staff in the given problem. However, after completing the tour, an ant need to release pheromone on its visited path to guide other ants. Consider our warehouse as a weighted graph where vertices are all shelves(including the door - shelf 0) and the weight of edge(i,j) is the istance from shelf i to j. When an ant is at shelf i, it will choose the next shelf to visit based on a probability function. The probability for an ant to go from shelf i to j is calculated through a heuristic function and the pheromone concentration on edge (i,j). After every ants complete its tour, we move to the next iteration where each ant will start a new tour again. The algorithm stops after a number of iterations.
Pseudo code for our ACO algorithm:

initialization: define problem parameters and initialize pheromone
while $iteration \leq threshold$ **do**

for ant in ants **do**
$cost = 0$
$kpi = initial\_kpi$
$visited[shelf] = False \, \forall \, shelf$ in $shelves$
$current\_shelf = 0$ and mark it as visited
while true **do**
Start from $current\_shelf$
Calculate the probability for every edge from $current\_shelf$
Choose the $next\_shelf$ to visit based on those probabilities
Update $current\_shelf = next\_shelf$ and mark it as visited
Update the $cost$, the remaining kpi
if reach goal **then**
Update the $best\_path$ and min_cost if $cost \leq min\_cost$
break the loop
end if
end while
Update the added pheromone on edges visited by this ant
end for
Update the pheromone concentration on every edges
$Iteration++$
end while
return $best\_path$

4.2.1 Initialization

We have to define some hyperparamters for ACO algorithm:

- α and β: two constants to balance the importance of pheromone trails and the importance of heuristic information.
- Pheromone evaporation rate p: a constant from 0 to 1 to control the pheromone concentration.
- Pheromone update strength Q: a constant to control the amount of pheromone deposited by an ant on edges of its path

We initialize the pheromone concentration on every edge(i,j) $R_{ij} = \frac{1}{d_{ij}} \, \forall \, 0 \leq i, j \leq M$

4.2.2 The pheromone update

After an ant k complete its tour, the added pheromenon to an edge(i,j) that it has traversed is calculated by this formula:

$$\Delta R_{ij}^{k} = Q/cost$$

After all ants completing their tour, we update the total pheromenon on every edges of the graph. Here, different from basic ACO, we also take advantage of elitist ant system (EAS) which is inspired by the paper[2]. In each iteration, the best path found within that

iteration receives an extra pheromone released by an elite ant. Our formula for the pheromone concentration update on edge (i,j) at the end of an iteration is:

$$R_{ij} = (1-p)R_{ij} + \Delta R_{ij} + \epsilon L_{ij}$$

where

$$\Delta R_{ij} = \sum_{k \in ants} \Delta R_{ij}^{k}$$

$$\epsilon = 0.5$$

$$L_{ij} = \begin{cases} Q/cost & \text{if } \ edge(i,j) \ \in \ best\_path\_in\_that\_iteration \\ 0 & \text{otherwise} \end{cases}$$

4.2.3 Probability function

When an ant is at shelf i, it will choose the next shelf to visit by calculating all probability on every edges from i to other shelves. Here the formula for the probability for an ant going from shelf i to j:

$$P_{ij} = \begin{cases} X/Y & \text{if shelf j is not visited} \\ 0 & \text{otherwise} \end{cases}$$

where

$$X = R_{ij}^{\alpha} H_{ij}^{\beta}$$

$$Y = \sum_{j \in J} R_{ij}^{\alpha} H_{ij}^{\beta} \ , \quad J : \text{non-visited shelves}$$

here, $H_{ij} = 1/d_{ij}$ which is actually our heuristic information.

4.3 Experiments

Because metaheuristics are very complicated and there are many things needed to be tuned to optimize them, in the section of ACO and lS-SA, we introduce a sub-section called Experiments to draw out some of many experiments that we have carried out to gain a better algorithm.
After a lot of experiments, we have founded suitable values for the hyperparameter:

$$\alpha = 1 \ , \ \beta = 3 \ , \ Q = 10 \ , \ p = 0.5$$

The number of ants is set to $0.1 shelves\_num$.
Using the above configuration, we will show a comparisons between the algorithm with normal ants only and the algorithm using EAS. The comparisons is performed on the set of testcases T containing 5 testcases with the number of shelves equal to 10,25,50,100 and 200 and the number of products all equal to 10. They are all generated randomly.For each testcase in T, each algorithm has run 5 times (except for the 200 shelves testcase because of our limited time, each algorithm only run once). The best min cost and the average min cost produced by 2 algorithmz are used for comparison.

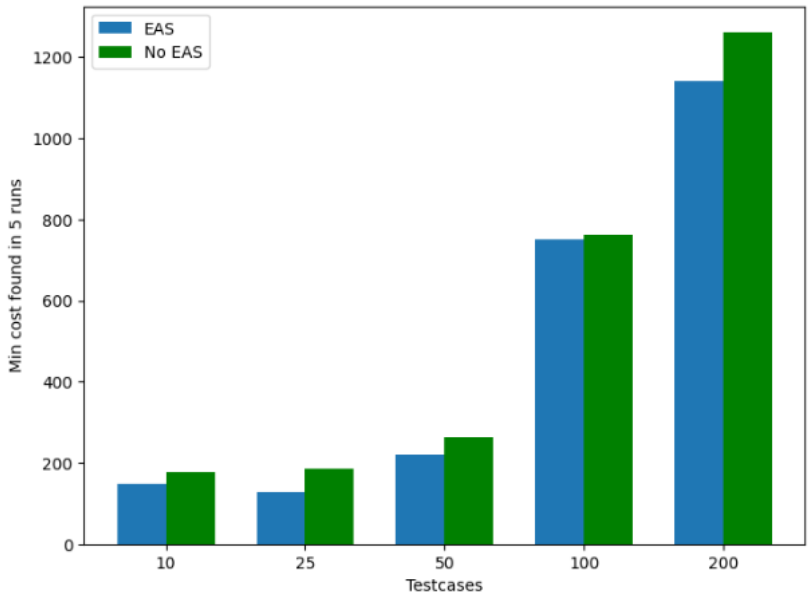

Figure 1: Comparison of min cost produced by 2 algorithms for 5 testcases

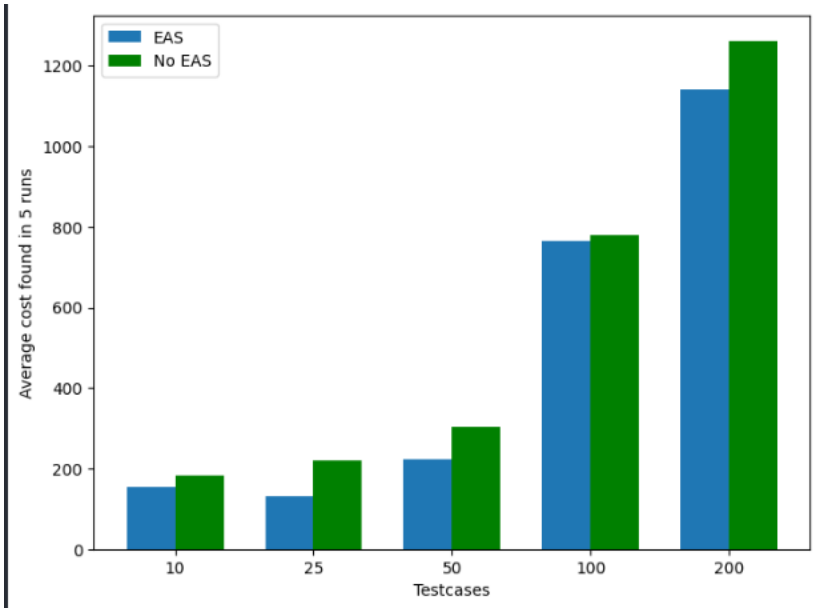

Figure 2: Comparison of average cost produced by 2 algorithms for 5 testcases

It is clear that the algorithm with EAS has improved the obtained answer.

5 Local Search - Simulated Annealing

5.1 Method

This algorithm is actually a simulated annealing(SA) algorithm combined with a local search(LS). The algorithm is inspired by the article [1]. Firsly, we create an initial valid solution by our greedy algorithm, apply the local search and set it to current solution. Then, a valid neighbour of the current solution will be generated. At this point, we do not immediately apply the test of SA. Instead, we use local search to improve the neighbour first. Then, the accept/reject test is applied. The process is repeated until a threshold is reached

initialization: set temperature T,cooling rate α

current_solution=local_search(*greedy_solution*)

while $iteration \leq threshold$ **do**

new_solution=make_neighbour(*current_solution*)

new_solution=local_search(*new_solution*)

$\Delta = new\_cost - current\_cost$

if $\Delta \leq 0$ **then**

$current\_solution = new\_solution$

$current\_cost = new\_cost$

Update the *best_solution* if the *new_solution* is better

else

Calculate $p = e^{-\Delta/T}$

Generate a random number k betweeen 0 and 1

if $k < p$ **then**

Update the solution and cost like above

end if

end if

$iteration++$

end while

Return *best_solution*

5.1.1 k-opt

K-opt local search is a famous technique for TSP. In this local search, the modification on a solution to create a neighbour is done through a k-opt move. Given a solution in tsp, it will choose k edges to remove, leaving disconnected segments . Then, these segments will be connected in different way to form a new solution.

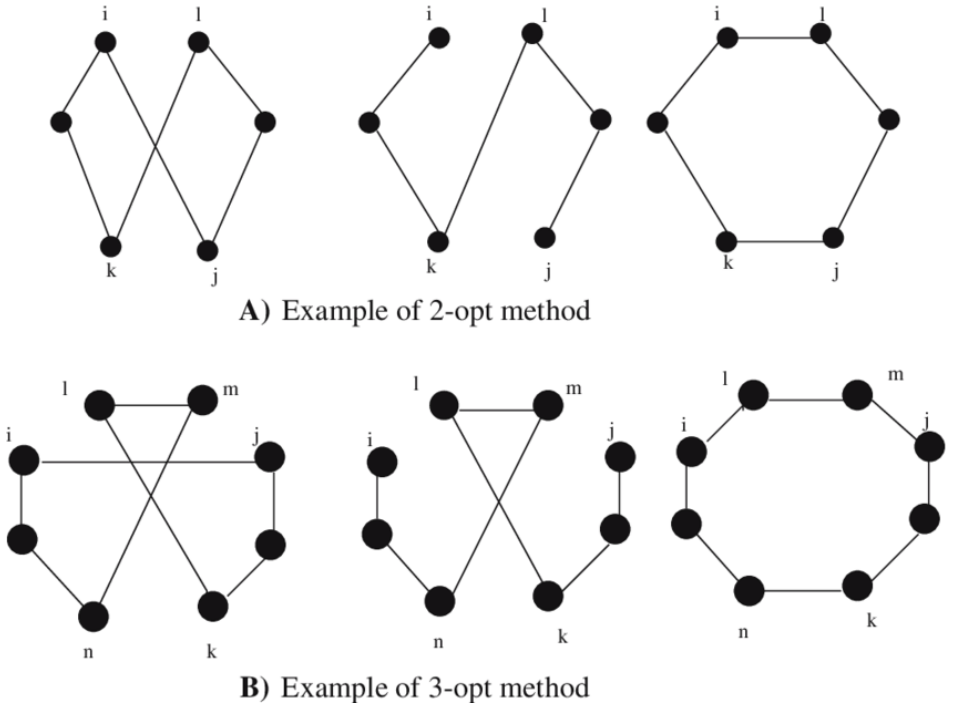

Figure 3: Example-of-2-opt-and-3-opt

5.1.2 Approach for generating neighbours

In TSP, generating good neighbours from a valid solution is not complicated since all valid solution must be a path of n cities. One can swap position of 2 random cities, or do a k-opt move. In this problem, generating good neighbours to provide a good landscape for searching is much harder since we do not know the number of shelves in the shortest path. Suppose we have a solution with k shelves, it is commnonly not good to only perform operations on only those shelves because many better solutions may lies in a combination of other shelves. However, our group has come up with an interesting approach. Let D be the set of shelves that not in current solution s, we will have a total of 4 operations:

- Replace: replace 1 shelf in the current solution by a shelf in D
- Add and replace: add 1 shelf from D to s and replace 1 shelf of s by a shelf from D
- Remove: remove 1 shelf from s
- 3-opt: perform a 3-opt move on s

A random number t between 0 and 1 is generated and based on the value of t, one of 4 above operations will be operated on s. Here, we make a asumption that the number of shelfs in the best solution cannot be larger than 1.25 times of that in the greedy solution. Hence, if the number of shelfs in the current solution exceeds that, we will perform the remove medthod only.

$n = shelves\_num\_in\_s$

$l = shelves\_num\_in\_greedy\_solution$

if $n \leq 1.25l$ **then**

  if $t \leq 0.25$ **then**

    replace(s)

  else if $t \leq 0.5$ **then**

    add_and_replace(s)

  else if $t \leq 0.75$ **then**

    remove(s)

  else 3-opt(s)

  end if

else remove(s)

end if

It is clear that among our above operations, the first three methods do not guarantee to provide a valid solution. Therefore, after a neighbour is generated, it is checked to see if it is valid or not, if not, we perform the function make_neighbour again until we got a valid neighbour. It is guaranteed that after a number of times calling the function make_neighbour we will get a valid solution because of the appearance of the 3-opt method.

5.1.3 Local search

The local search we use to improve the neighbour solution here is 2-opt local search. We think this is a good combination with the make_neighbour function because the changes applied on current solution here are quite diverse. This is to say, if the modification of the current solution after applied local search is similar to that of make_neighbour function, it is likely that the state would go back and forth, which is not good because the algorithm can stuck at a local minima.

5.2 Experiments

We want to show the effect of choosing a good cool down rate to assure good exploration of the algorithm. Here, we set T=100 and experiment with different cool down rates. The experiments is carried out on a 100 shelves testcase and the number of iteration is 20000. The below graphs illustrate the current cost of the current solution in each iteration to see the influence of the cooling rate to the algorithm in accepting/rejecting a new solution that is worse than the current one.

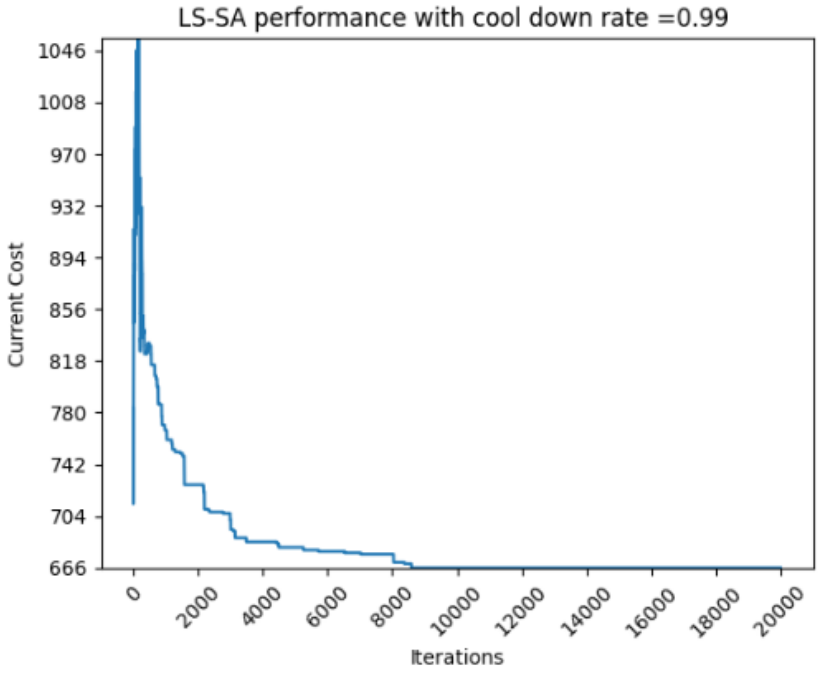

Figure 4

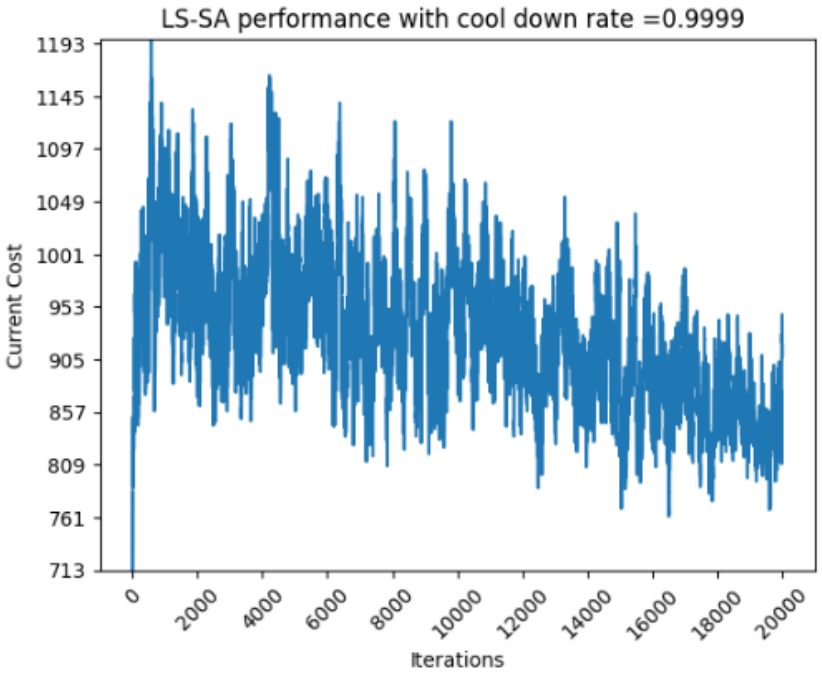

Figure 5

As can be seen from the graph with the cooling rate=0.99, at the very first number of iterations, when T is still high, the algorithm easily accepts worse solution (current cost going up). T is cooling down gradually and followed by that, current cost tends to go down. However, with cool down rate=0.99, T become too low after thoudsands of iterations and

the algorithm become a normal hill climbing search so it stucks at a local minimum after iteration around 8000.

We can avoid that by letting cool down rate very high like 0.9999. But it is apparent from the graph that very high cool down rate make the algorithm too random , the minimum cost in the graph is even the cost at the first iteration which means the algorithm do not even improve the initial solution.

Therefore, we have come up with an initiative called reheating: after 200 iterations without finding a better neighbour than the current solution, increase T by 20. This technique helps the algorithm avoid stucking and too much randomness.

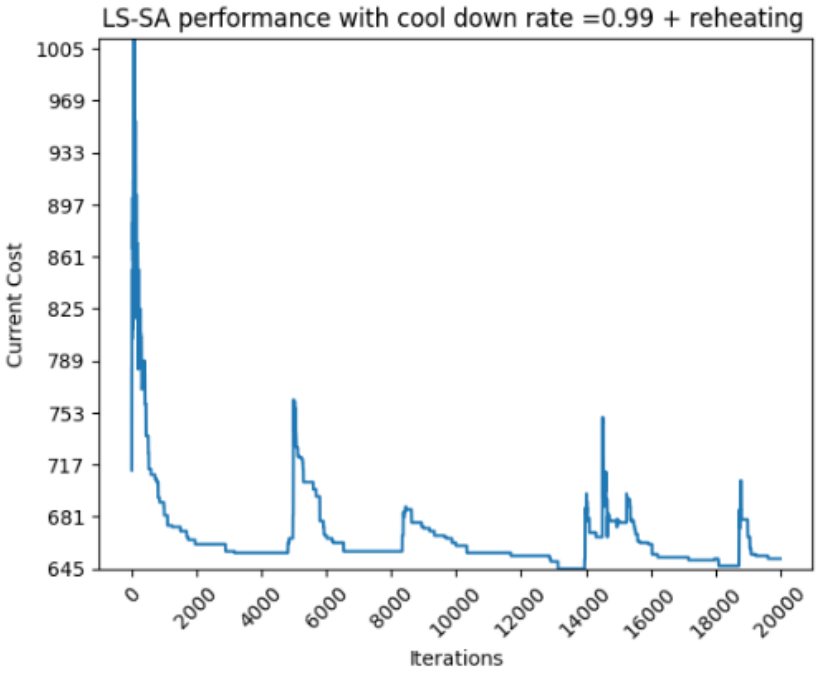

Figure 6

Part III
Algorithms Comparison

Comparison of our 6 algorithms is carried out on a randomly generated 5 testcases in which there are 10,25,50,100,200,500 shelves. The number of products in all testcases is 10. The number of ants in ACO is $0.1 \times shelves\_num$ and the number of iteration is $200000/shelves\_num$. The number of iterations in LS-SA is $2000000/shelves\_num$.

Here the result of the comparison. Note that N/A means the algorithm runs too long that cannot produce result.

	10	25	50	100	200	500
BB	148	121	N/A	N/A	N/A	N/A
IP	148	121	198	N/A	N/A	N/A
CP	148	121	198	N/A	N/A	N/A
GREEDY	217	172	292	783	1051	2525
ACO	148	128	220	752	1067	2734
LS-SA	148	122	218	636	896	2197

Figure 7: Length of obtained path comparison

	10	25	50	100	200	500
BB	0.03	65.1	N/A	N/A	N/A	N/A
IP	0.2	2.4	43.3	N/A	N/A	N/A
CP	0.2	2.9	35.4	N/A	N/A	N/A
GREEDY	0	0	0	0.001	0.001	0.007
ACO	6.8	15.5	32.5	83.7	239.2	1362.9
LS-SA	25.2	34.3	56.2	103.9	108	355.1

Figure 8: Runtime(seconds) comparison

From that, we can summary the weakness and strength of each algorithm in the table below:

	Advantages	Disadvantages
BB	Exact solution	Run too slow, can only solve problem with very smalldataset
IP	Exact solution, run faster than BB, recommended for small testcase (<= 80 shelves)	Can not solve problem with large dataset
CP	Similar to IP	Similar to IP
Greedy	Run very fast, produce fairly good result, suitable for any size of testcase	In many cases, the produced result is far from the global optimum
ACO	Produce fairly good result	Run quite slow, prone to stuck at local optimum
LS-SA	Run fast, produce better result than greedy	May need to run the algorithm several times to produce good results due to its randomness

Figure 9: Algorithms weakness and strength summary

Part IV
Conclusion

This mini project has provided us a lot of knowledge about algorithms design to solve an optimization problem. We can conclude that when implementing an algorithm, there is always a trade off. We can only either choose a fast one producing acceptable result or a very slow one producing excellent outcome. For these np-hard problems, there is no such algorithm that is able to run within seconds for a large testcase and still give very good answer. According to the size of data, we can pick a suitable algorithm. Among an unlimited number of algorithms out there, we think greedy is a diamond in optimization. In real-life scenarios, we believe that a simple greedy algorithm or an algorithm that is improved from greedy is an extremely efficient tool that people should think of first when dealing with any np-hard problems.

References

[1] Olivier Martin and Steve Otto. Combining simulated annealing with local search heuristics. *Annals of Operations Research*, 63, 04 1999.

[2] Sorin Negulescu, Constantin Oprean, Claudiu Kifor, and Ilie Carabulea. Elitist ant system for route allocation problem. 01 2008.