CSE 424 OPTIMIZATION REPORT OF ALL ASSIGNMENTS

Fatih Selim YAKAR - 161044054

- ## Verbal Problem Definition

There is warehouse with n plastic cases to store k different products. Also there are m orders with k different products. In this context, the cases that will meet the total order product values will be selected. When choosing these cases, the aim is to choose the least incremental and to meet all product needs.

The problem is similar to the knapsack problem in general. In this context, a solution can be established by considering the sum of orders as a knapsack and cases as items thrown into it.

- ## Problem Definition As An Optimization Problem

There are 4 main definitions to describe an optimization problem:

1. *Decision Variables:*

$$x_1, x_2, x_3, x_4 \ldots\ldots x_n \ (the\ index\ of\ the\ first\ to\ nth\ choosen\ case)$$

2. *Domain:*

$$D_{1\ldots n} = \{\ 0(\emptyset), 1,2\ldots\ldots\ldots n\ \}$$

If the decision variable is 0 or null, this indicates that no case was selected. In other cases (1 .... n) it shows the index of the selected case.

3. *Constraints:*

$$(1)\ x_i \neq x_j$$

The indexes selected by Decision varibles should not be the same.

$$(2)\ \left(\sum_{i=1}^{m} i_{p_{value}} \leq \left[\sum_{j=1}^{n} x_{j_{p_{value}}}, x_j \neq 0\ \right]\right)_{for\ all\ p=1\ldots..k\ (products)}$$

For all products, the sum of all orders must be equal to or less than the sum of the selected cases.

4. *Objective Function:*

$$minimize \sum_{p=1}^{k} \left(\left[\sum_{j=1}^{n} x_{j_{p_{value}}}, x_j \neq 0 \right] - \sum_{i=1}^{m} i_{p_{value}} \right)$$

For all products, the sum of the difference between the sum of the selected cases and the sum of all orders should be minimum.

- ## General Methods And Classes Used In Solving The Problems

1. Product_List Class

It contains a vector(list) of integer product values in orders and cases, and methods that manipulate and manipulate that list.

Methods:

Product_List(): Default constructor that creates a Product_List object.

Product_List(int array[],int product_size): Initializes the private vector from an integer array.

Product_List(vector<int> vector)): Initializes the private vector from an integer vector.

void add_product(int value): Adds a new integer value to the private vector.

void print_products(): Prints the private product_list.

vector<int> get_vector(): Returns the private vector.

2. Orders Class

Maintains a list of orders within the private Product_List vector. It manipulates and processes this list with its methods.

Methods:

Orders(): Default constructor that creates a Orders object.

Orders(vector<Product_List> vector): Initializes the private vector from an Product_List vector.

void add_order(Product_List order): Adds a new Product_List to the private vector.

void print_orders(): Prints the private order list.

vector<int> return_sum_value(): Returns vector that includes the summation of the order list's product values

bool is_lower(vector<int> vect): Checks whether each element of the vector given as a parameter is greater than or equal to each element of the vector returned as the result of return_sum_value. If so it returns true, otherwise it returns false.

vector<Product_List> get_vector(): Returns the private vector.

3. Cases Class

Maintains a list of orders within the private Product_List vector and case_nums vector. It manipulates and processes this lists with its methods.

Methods:

Cases(): Default constructor that creates a Cases object.

Cases(vector<Product_List> vector): Initializes the private vector from an Product_List vector.

void add_case(Product_List the_case): Adds a new Product_List to the private vector.

void print_cases(): Prints the private case list.

vector<int> return_sum_value(vector<int> indexes): Returns vector that includes the summation of the case list's choosen product values.

int get_case_list_size(): *Returns the cases object's vector size.*

vector<Product_List> get_vector(): *Returns the private case_list vector.*

vector<int> get_vector_case_nums(): *Returns the private case_nums vector.*

void sort_by_order_sum(vector<int> order_sum): *Sorts the case_list and case_nums by looking value.*

void sort_by_order_sum_absolute(vector<int> order_sum): *Sorts the case_list and case_nums by looking absolute value.*

void sort_by_bound(vector<int> order_sum): *Sorts the case_list and case_nums by looking bounds.*

void find_difference_sum(int index,vector<int> order_sum): *Finds difference of case sum and order sum's sum.*

void find_difference_sum_absolute(int index,vector<int> order_sum): *Finds difference of case sum and order sum's absolute sum.*

4. Algorithm Class

Includes the general algorithm features as parent class

Methods:

Algorithm(): Constructor of the Algorithm class.

Algorithm(int test_index): Constructor of the Algorithm class.

void set_test_index(int index): *Setter of test_index variable.*

int get_test_index(): *Getter of test_index variable.*

void read_and_setup(string filename): It reads the test file. It adds the information it reads to the cases and orders vectors in the object. At the end of the function, there are cases and orders vectors with size as the number of tests.

int rng_zero_between_one(): *Returns the arbitrary number between 0-1.*

Brute Force Algorithm (HW1)

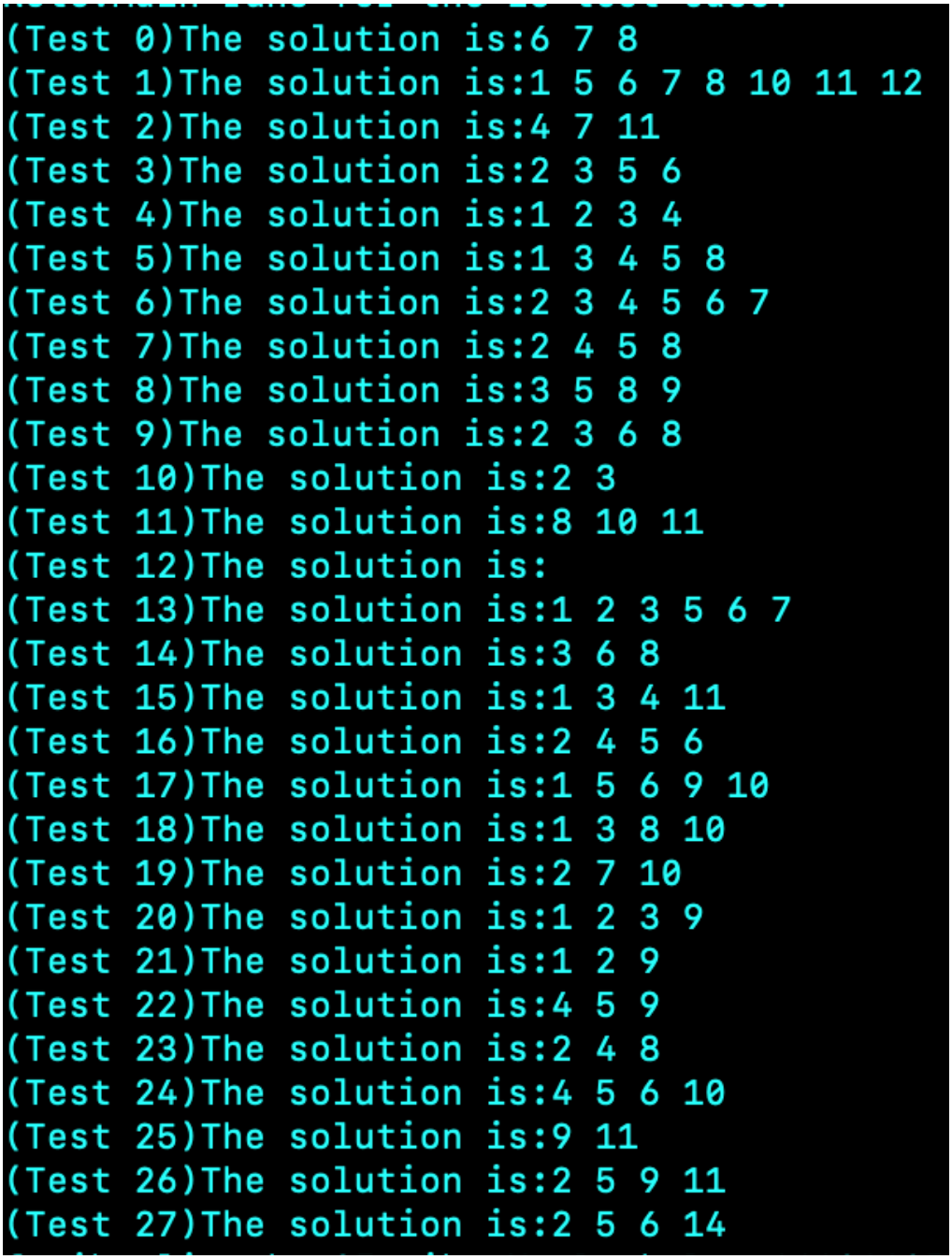
First, it is necessary to add them all in order to make a single total. The next process requires a brute force approach. So try any combination of the selection of cases. It is tried to find the smallest sum of the differences of product values in this combination. This means approximately C (n, 0) + C (n, 1) + ... + C (n, n) trials, with n = the number of cases.

- ### Pseudocode of Brute Force Algorithm

```
function brute_force_algorithm(int test_num)
     for i=1 to case_num[test_num]
          wrapper_combination(..)
```

```
        end for

        for i=0 to min_indexes.size()
             ++min_indexes[i]
        end for

        print solution
    end function

    function wrapper_combination(int set_index, int size, int
    combination)
        initialize array[size]
        for i=0 to size
             array[i]=i
        end for
        initialize data[combination]

        combination_process(..) //tries all of combinations
    and returns min valued
    end function
```

- **Tests Given Text File (HW1)**



Greedy Algorithm (HW2)

For the greedy solution, I applied the following target frame:

```
Greedy Algorithm ( a [ 1 .. N ] ) {
      solution = Ø
      for i = 1 to n
            x = select (a)
            if feasible ( solution, x )
                  solution = solution U {x}
      return solution
}
```

Our goal was actually to select the cases that meet the order with the least waste. To achieve this, I first listed the cases obtained after the file was read according to greedy decision. Then I collected the cases by taking the cases from the sequenced array so that there are as many loops as the total number of cases. As soon as the result of the collected cases corresponds to the order, I stopped the loop and returned that collected case list. In fact, my first ordering of cases was to get the smallest value in each loop sorted as a greedy decision. So the purpose of this order was to make a greedy decision on each loop and add that option to the selected case list.

In order to achieve this greedy decisions, I used 2 approaches, which means that I use 2 greedy algorithms for finding comparison values.

Approach 1: If order is (6,4) and case is (10,0), we find the difference (case-order) one by one and take the sum of these differences. For this case, 10-6 + 0-4 = 0.

Approach 2: If order is (6,4) and case is (10,0), we find the difference and absolute value (case-order) one by one and take the sum of these differences. For this case, abs(10-6) + abs(0-4) = 8.

Pseudocode For Greedy Algorithms

```
greedy algorithm(order,case_list[1…n]){
      find the summation vector of order
      Sort the case list by using approach1 or approach2

      İnitialize the sum vector
      İnitialize the choice vector
```

```
        For i=1 to n
            Sum+=case_list[i]
            Choice.push_back(choosen case indexes)
            İf meets the order
                Break
        Return choice
}
```

Also I calculated the error deviation like this:

İnitialize Total error
For(i=0 to n)
Total error+=(Choosen cases summation product i – orders summation product i) / orders summation product i

Dynamic Programming Algorithm (HW2)

In dynamic programming, it is necessary to divide it into sub-problems with a recursive approach. In this way, it is necessary to reach a solution by solving sub-problems and keeping them in the dp table. In this context, each sub-problem would be saved to the table and used from here when recalculated, and unlike brute-force, it would not go around all the solution possibilities without the need for recalculation.

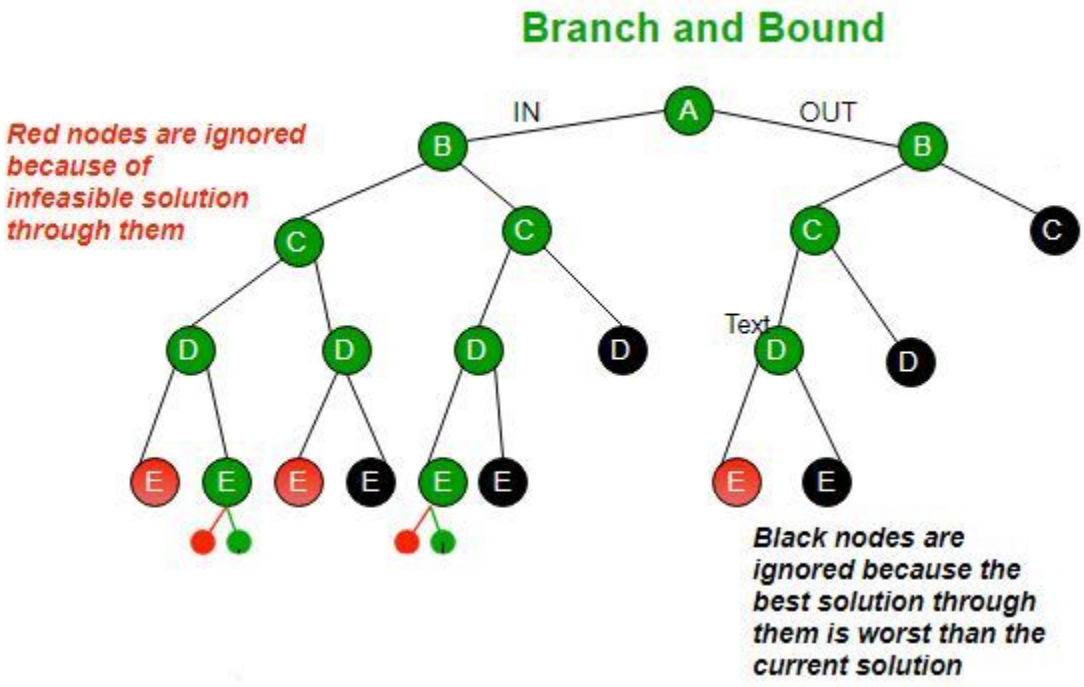
In order to achieve this, I tried to solve it over the knapsack problem, which is the closest problem to our problem. I tried to do an abstraction for this. This abstraction included: Thinking of the products in case and order as a single value, not a lot, and processing accordingly. I was going to keep this abstracted value as a vector.

Although I came close to generating solutions in this simple state, **I could not do dynamic programming** because I had difficulty in applying to the actual problem. The solution would be in this framework if I could:

- Relating a solution to a larger instance to solutions of some smaller instances
- Solve smaller instances once
- Record solutions in a table
- Extract solution to the initial instance from the table

Branch and Bound Algorithm (HW2)

An approach similar to the greedy algorithm was developed in branch-and-bound. Bounds are found in terms of the distance from the totals to orders, similar to that in greedy. Solutions were eliminated according to these bound and feasible. Finally, a solution was found.



Pseudocode For Branch and Bound Algorithms

```
Branch-and-bound(order,case_list[1…n]){
     İnitialize the vectors
     Sort the case list by using bound

     For i=1 to n
          Sum+=case_list[i]
          Choice.push_back(indexes)
          İf meets the order
               Return choices

}
```

- **Tests Given Text File (HW2)**

```
Greedy Algorithm2
(Test0-Greedy2)The solution is:4 2
Error:0.965954
(Test1-Greedy2)The solution is:2 7 1 12 4 6 9 10 3 5
Error:7.93844
(Test2-Greedy2)The solution is:6 7 11
Error:0.183694
(Test3-Greedy2)The solution is:6 7 4 3
Error:8.19744
(Test4-Greedy2)The solution is:8 7 6 5 2
Error:5.00928
(Test5-Greedy2)The solution is:9 7 4 8
Error:40.5415
(Test6-Greedy2)The solution is:4 3 7 2 5 1 6
Error:34.5794
(Test7-Greedy2)The solution is:8 3 6 7
Error:44.4944
(Test8-Greedy2)The solution is:6 9 1 2
Error:42.9752
(Test9-Greedy2)The solution is:1 6 9 10
Error:55.7129
(Test10-Greedy2)The solution is:3 4 2
Error:375.177
(Test11-Greedy2)The solution is:8 3 11 5 10
Error:274.955
(Test12-Greedy2)The solution is:4 3 1 2 6 5 7
Error:17.1875
(Test13-Greedy2)The solution is:9 10 1 2
Error:111.292
(Test14-Greedy2)The solution is:6 8 3
Error:161.008
(Test15-Greedy2)The solution is:7 11 4 1 3
Error:217.772
(Test16-Greedy2)The solution is:2 6 7 4 3
Error:228.023
(Test17-Greedy2)The solution is:7 6 1 3
Error:105.81
(Test18-Greedy2)The solution is:9 3 2 10
Error:191.66
(Test19-Greedy2)The solution is:2 7 3 8
Error:237.795
(Test20-Greedy2)The solution is:3 2 1 4 8
Error:367.24
(Test21-Greedy2)The solution is:8 7 2 6 9
Error:228.67
(Test22-Greedy2)The solution is:5 8 9 1
Error:275.864
(Test23-Greedy2)The solution is:4 7 8 3
Error:243.537
(Test24-Greedy2)The solution is:4 10 2 5 8
Error:244.136
```

```
Greedy Algorithm1
(Test0-Greedy1)The solution is:6 5 3 8 7
Error:1.72945
(Test1-Greedy1)The solution is:11 8 5 3 10 9 6 4 12 1 7
Error:7.33773
(Test2-Greedy1)The solution is:2 9 3 8 5 1 10
Error:1.28304
(Test3-Greedy1)The solution is:2 8 1 5 3 4
Error:8.46665
(Test4-Greedy1)The solution is:3 4 1 2
Error:1.27829
(Test5-Greedy1)The solution is:1 6 5 3 2 8 4
Error:42.2885
(Test6-Greedy1)The solution is:6 1 5 2 7 3 4
Error:34.5794
(Test7-Greedy1)The solution is:4 2 1 5 3 8
Error:40.1304
(Test8-Greedy1)The solution is:8 4 5 7 2
Error:31.7694
(Test9-Greedy1)The solution is:3 5 4 2 11 8 10
Error:71.2615
(Test10-Greedy1)The solution is:3 4 2
Error:375.177
(Test11-Greedy1)The solution is:8 3 11 5 10
Error:274.955
(Test12-Greedy1)The solution is:7 5 6 2 1 3 4
Error:17.1875
(Test13-Greedy1)The solution is:4 3 6 5 7 1 9 2
Error:97.9026
(Test14-Greedy1)The solution is:6 3 8
Error:161.008
(Test15-Greedy1)The solution is:4 11 7 3 1
Error:217.772
(Test16-Greedy1)The solution is:4 6 2 7 3
Error:228.023
(Test17-Greedy1)The solution is:5 9 10 2 3 1
Error:87.7323
(Test18-Greedy1)The solution is:8 1 6 3 10
Error:137.705
(Test19-Greedy1)The solution is:7 2 3 8
Error:237.795
(Test20-Greedy1)The solution is:1 3 2 4 8
Error:367.24
(Test21-Greedy1)The solution is:2 8 7 10 6 9
Error:315.02
(Test22-Greedy1)The solution is:7 6 5 8 9 1
Error:329.544
(Test23-Greedy1)The solution is:8 7 4 3
Error:243.537
(Test24-Greedy1)The solution is:6 5 10 4
Error:105.124
```

```
BB Algorithm
(Test0-BB)The solution is:4 2
(Test1-BB)The solution is:2 7 1 12 4 6 9 10 3 5
(Test2-BB)The solution is:6 7 11
(Test3-BB)The solution is:6 7 4 3
(Test4-BB)The solution is:8 7 6 5 2
(Test5-BB)The solution is:9 7 4 8
(Test6-BB)The solution is:4 3 7 2 5 1 6
(Test7-BB)The solution is:8 3 6 7
(Test8-BB)The solution is:6 9 1 2
(Test9-BB)The solution is:1 6 9 10
(Test10-BB)The solution is:3 4 2
(Test11-BB)The solution is:8 3 11 5 10
(Test13-BB)The solution is:9 10 1 2
(Test14-BB)The solution is:6 8 3
(Test15-BB)The solution is:7 11 4 1 3
(Test16-BB)The solution is:2 6 7 4 3
(Test17-BB)The solution is:7 6 1 3
(Test18-BB)The solution is:9 3 2 10
(Test19-BB)The solution is:2 7 3 8
(Test20-BB)The solution is:3 2 1 4 8
(Test21-BB)The solution is:8 7 2 6 9
(Test22-BB)The solution is:5 8 9 1
(Test23-BB)The solution is:4 7 8 3
(Test24-BB)The solution is:4 10 2 5 8
fatihselimyakar@Fatihs-MacBook-Pro HW2 %
```

VNS Algorithm (HW3)

For the VNS solution, I applied the following target frame:

```
Procedure VNS
  Define neighborhood structures N_k (k=1,...,k_max)
  Generate initial solution s Є S
   while stopping condition is not met do
   k ← 1
   while k ≤ k_max do
       s' ← Shake(s), s' Є N_k (s)
       s" ← LocalSearch(s'), s" Є S
       if (Fitness(s") < Fitness(s))
             s ← s"
             k ← 1
       else
                k ← k+1
   end-while
  end-while
End-Procedure
```

I chose VNS because it has the ability to mix up the solution more and it has local search than RVNS. Against VND, it was spending less computational time because it was not looking for the best.

In this warehouse problem, I think it like Orienteering Problem. So I represent the this warehouse problem like a permutation problem. Although this problem is not permutation problem,I changed the chosen case by changing case selecting order with neigborhood structures. Also I define the feasibility limit for changing and inserting line.

After the neighborhoods definition. I have defined 4 neighborhoods in total. These were insert, exchange, insert path, exchange path neighborhoods. I used path exchange and path insert here, since I need to mix more in the shake function. I used insert and exchange here because it needed to mix less in the local search function. After defining the neighborhoods and shake, local search function, it was only left to question the initial solution and its adequacy. For the Initial solution, it was necessary to create a permutation with the solution, I simply did it as {0,1,2,3} for a 4-element problem. I decided the adequacy according to the smalling of the increasing amount as in the objective function.

Problem Representation for this problem:

Orders		
	Product 1	Product 2
Order 1	2	0
Order 2	2	4
Order 3	2	0
Order Total	6	4

Cases		
	Product 1	Product 2
Case 1	10	0
Case 2	0	21
Case 3	6	3
Case 4	8	4

As the permutation problem these are the solutions:

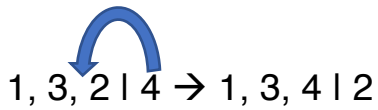
- 1, 2, 3, 4 → Initial solution for this problem.

- 4 | 2, 3, 1 → The feasibility limit is 1 in this permutation case.
- 1, 3, 2 | 4 → The feasibility limit is 3 in this permutation case.
- 3, 4 | 1, 2 → The feasibility limit is 2 in this permutation case.
- .
- .
- .
- 4! permutations

Neighborhoods

- **Insert Neighborhood**

Finds randomly upper bound in an index greater than or equal to the feasible limit. Finds the feasible limit in a small index lower bound. Then it inserts the value in the upper bound into the index in the lower bound.



- **Exchange Neighborhood**

Finds randomly upper bound in an index greater than or equal to the feasible limit. Finds the feasible limit in a small index lower bound. Then it exchange the value in the upper bound and value in the lower bound.

1, 3, 2 | 4 → 1, 4 | 2, 3

- **Insert Path Neighborhood**

  Sets an index from the indexes randomly above and below the feasible limit. Insert the upper bound index value to end of the indexes path into the lower bound index.

1, 4 | 2, 3 →1, 2 | 3, 4

- **Exchange Path Neighborhood**

  Sets an index from the indexes randomly above and below the feasible limit. Exchange the upper bound index value to end of the indexes path into the lower bound index.

1, 4 | 2, 3 →2, 3 | 1, 4

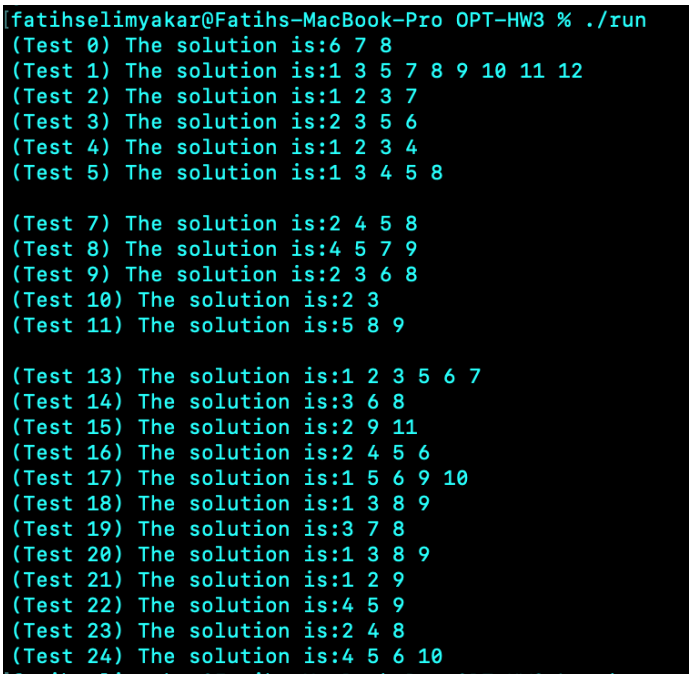
Pseudocodes of the VNS algorithm

```
void VNS_algorithm(){
     Initialize the t_max,k,k_max,feasible_limit,difference.
     Initialize the initial_solution<int> and indexes<int>.
     Create an initial solution by looking size.
     Determine the initial solution's feasible limit.
     Repeat the following sequence until the t_max==t.
          k=0
          while k<k_max
               initialize new_limit
               shake(k,initial_solution,new_limit)
               local_search(k,initial_solution,new_limit)
               if new_solution's waste<old_solution_waste
                    k=0
                    initial_solution=new_solution
                    feasible_limit=new_limit
               else
                    ++k
               end if
          end while
```

```
     end repeat

     print initial_solution

}


vector<int>
shake(neigborhood,initial_solution<int>,&feasible_limit){
     initialize changed_solution<int> vector
     if neighborhood == 0
          changed_solution = insert_path_neighborhood
     else if neighborhood == 1
          changed_solution = exchange_path_neighborhood
     end if

     feasible_limit = new_feasible_limit

     return changed_solution
}


vector<int>
local_search(neigborhood,initial_solution<int>,&feasible_limit){
     initialize changed_solution<int> vector
     if neighborhood == 0
          changed_solution = insert_neighborhood
     else if neighborhood == 1
          changed_solution = exchange_neighborhood
     end if

     feasible_limit = new_feasible_limit

     return changed_solution
}
```

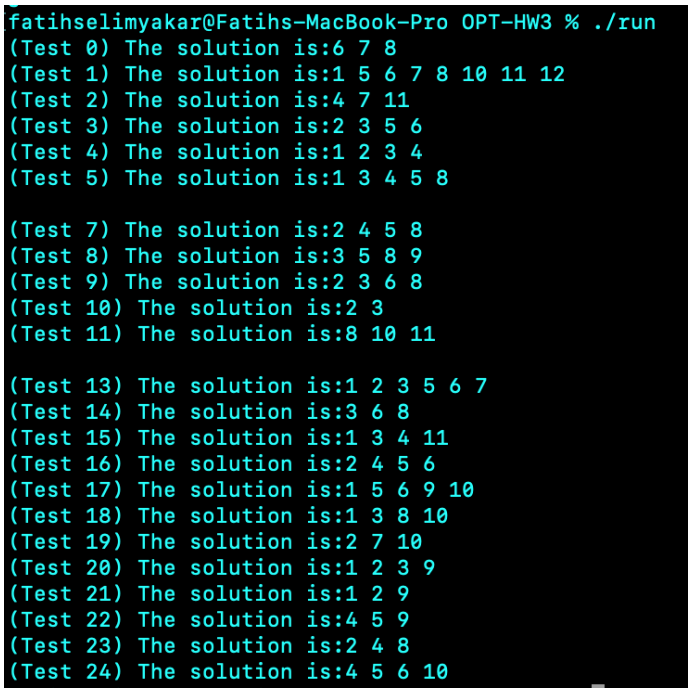
- **Tests Given Text File (HW3)**

  - **While t_max equal to 100**

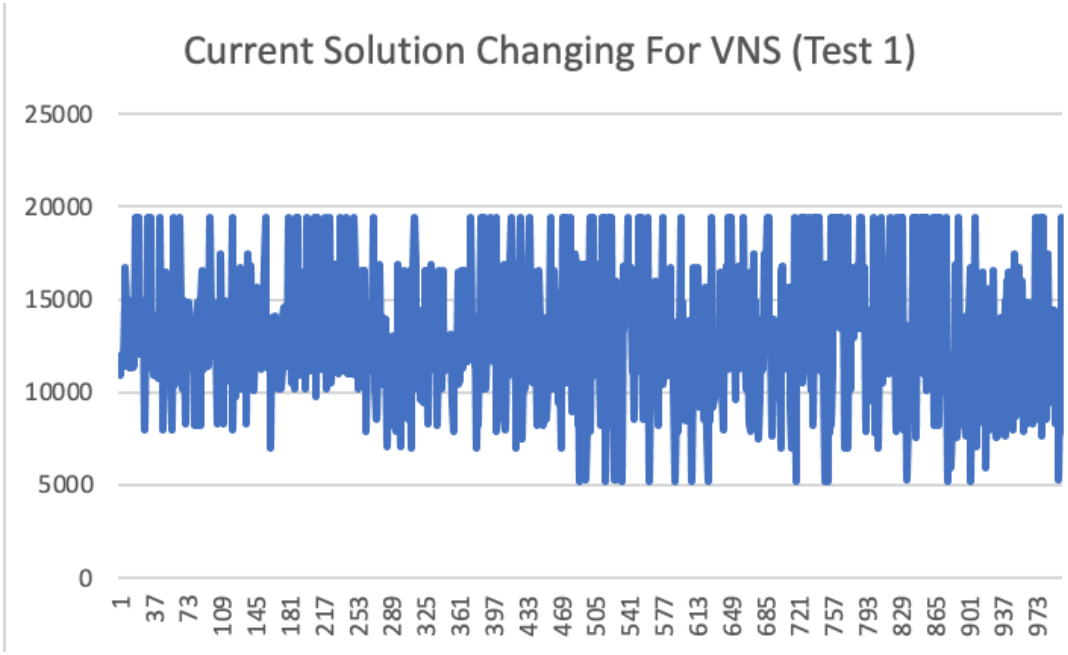


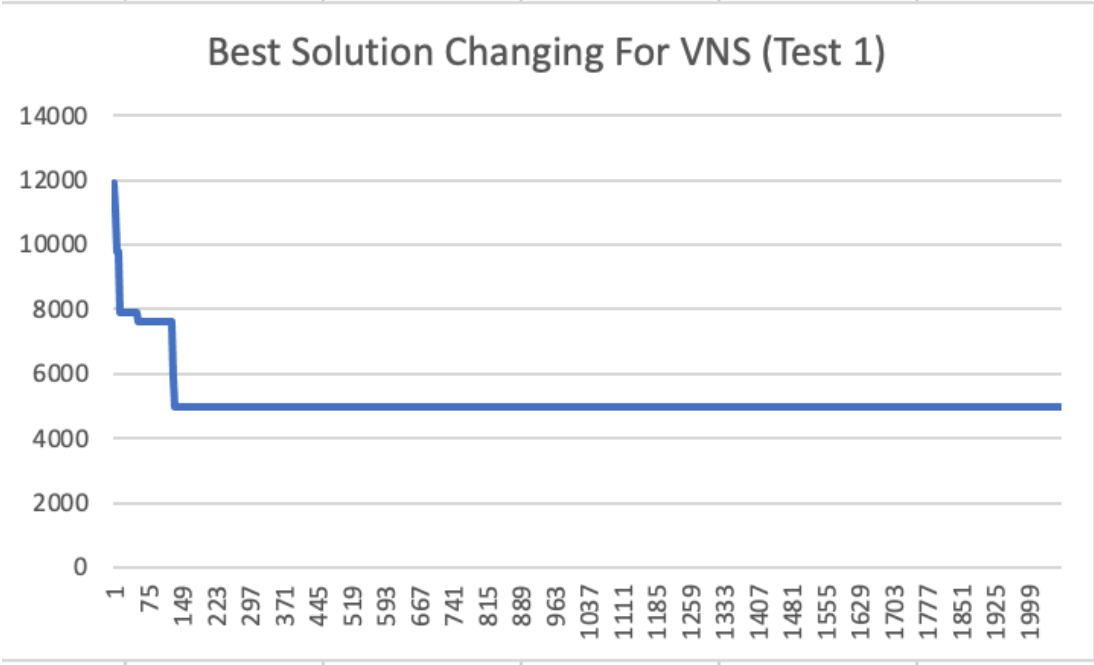
This test takes less time, but finds 7 non-optimal solutions in 23 trials.

  - **While t_max equal to 1000**



This test takes more time, but finds 23 optimal solutions in 23 trials.





Attention: The homework in the submission runs at t_max = 1000. It may take a long time for some tests.

Simulated Annealing Algorithm (HW4)

For the Simulated Annealing solution, I applied the following target frame:

```
procedure simulated annealing
begin
   t ← 0
   initialize T
   select a current point v_c at random
   evaluate v_c
   repeat
      repeat
         select a new point v_n
            in the neighborhood of v_c
         if eval(v_c) < eval(v_n)
            then v_c ← v_n
            else if random[0, 1) < e^((eval(v_n)−eval(v_c))/T)
                 then v_c ← v_n
      until (termination-condition)
      T ← g(T, t)
      t ← t + 1
   until (halting-criterion)
end
```

I chose only SA because the simplicity of SA was generally sufficient to solve this problem. Instead, the reheated SA or iterative SA that I will use would make the calculation more difficult by bringing more processing requirements on the simple SA that is already sufficient..

In this warehouse problem, I think it like Orienteering Problem. So I represent the this warehouse problem like a permutation problem. Although this problem is not permutation problem,I changed the chosen case by changing case selecting order with neigborhood structures. Also I define the feasibility limit for changing and inserting line.

For the Initial solution, it was necessary to create a permutation with the solution, I simply did it as {0,1,2,3} for a 4-element problem. I decided the adequacy according to the smalling of the increasing amount as in the objective function.

Problem Representation for this problem:

Orders		
	Product 1	Product 2
Order 1	2	0
Order 2	2	4
Order 3	2	0
Order Total	6	4

Cases		
	Product 1	Product 2
Case 1	10	0
Case 2	0	21
Case 3	6	3
Case 4	8	4

As the permutation problem these are the solutions:

- 1, 2, 3, 4 → Initial solution for this problem.

- 4 | 2, 3, 1 → The feasibility limit is 1 in this permutation case.
- 1, 3, 2 | 4 → The feasibility limit is 3 in this permutation case.
- 3, 4 | 1, 2 → The feasibility limit is 2 in this permutation case.
- .
- .
- .
- .
- 4! Permutations

Neighborhood

- **Exchange Neighborhood**

  Finds randomly upper bound in an index greater than or equal to the feasible limit. Finds the feasible limit in a small index lower bound. Then it exchange the value in the upper bound and value in the lower bound.

1, 3, 2 | 4 → 1, 4 | 2, 3

Pseudocodes of the SA algorithm

```
void SA_algorithm(){
     Initialize the
     t_max,k,k_max,feasible_limit,difference.bests_feasible_limi
     t,bests_difference,total_difference
```

```
     Initialize the
     initial_solution<int>,indexes<int>,best_solution<int> and
     temperature_indexes<int>

     Create an initial solution by looking size.

     Determine the initial solution's feasible limit.

     Finds the total_difference in case of selecting all cases
     and makes temperature.

     Temperature=total_difference

     Repeat the following sequence until the t_max==t.
          k=0
          while k<k_max
               initialize new_limit
               neighborhood_search(initial_solution,new_limit)
               if new_solution_waste<bests_difference
                    bests_difference=new_difference
                    bests_feasible_limit=new_feasible_limit
                    best_solution=new_solution
               end if
               if new_solution_waste<old_solution_waste
                    initial_solution=new_solution
                    feasible_limit=new_limit
               else if random(0,1)<exp((old_solution_waste-
                    new_solution_waste)/temperature)
                    initial_solution=new_solution
                    feasible_limit=new_limit

               end if
               ++k
          end while
          temperature=temperature*0.9
     end repeat

     print best_solution

}

vector<int>
local_search(neigborhood,initial_solution<int>,&feasible_limit){
     initialize changed_solution<int> vector

     changed_solution = exchange_neighborhood
```

```
        feasible_limit = new_feasible_limit

        return changed_solution
}
```

- **Tests And Hyper Parameters Given Text File (HW4)**

I tried to find the number of loop of the **inner and outer loop** by trying it several times. Since there are too many n and n ^ 2 complexity operations inside and outside, when both inner and outer loops return more than 100, it took an unexpectedly long time for middle level problems. To prevent this, I first set it to 100-100, but again after the 13th test, it was very slow and almost stopped. Finally, again for some problems, I got the best results and relatively better waiting time when the outer loop turns 100 times the inner loop 50 times, even if it takes too long.

As the **initial temperature**, I tried to find a value that will provide a figure between 0 and 1. I thought that this is the best maximum value of the objective function. In other words, when all cases are selected for the solution, I chose the amount of waste as temperature.

I have done many trials for the **cooling coefficient**. I first thought of a reduction based on continuous impact to reduce temperature. In other words, the formula is "temperature = temperature * cooling coefficient". Then I tried values between 0-1 for this temperature for the first 13 test cases. As a result of these tests, I got the following information:

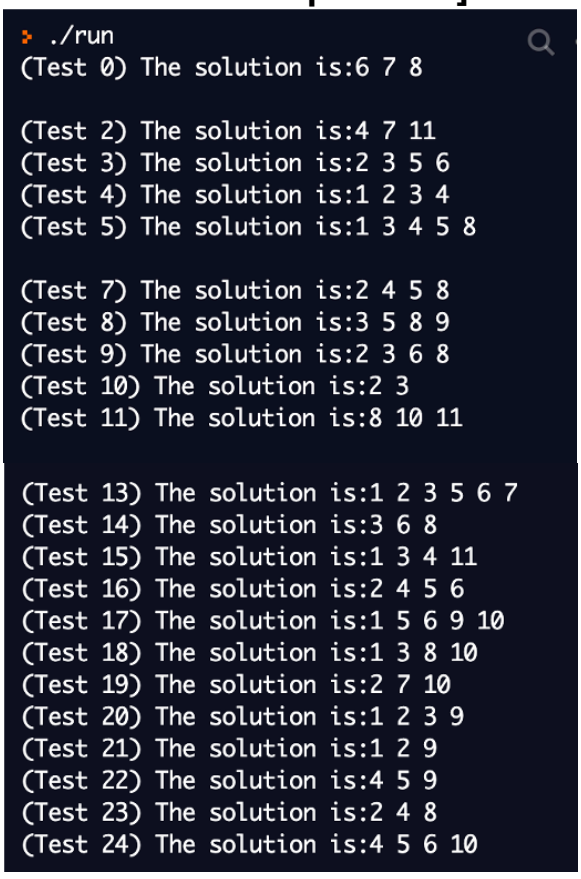
Cooling Coefficie	# of wrong result
0.1	2
0.2	1
0.3	3
0.4	4
0.5	1
0.6	0
0.7	0
0.8	0
0.9	0
0.99	0

As seen in the table, the number of erroneous results between the values of 0.6-1 was 0. But in order to be able to choose between them, this time I tried to operate with a 0.6-1 cooling coefficient for the first 24 tests.

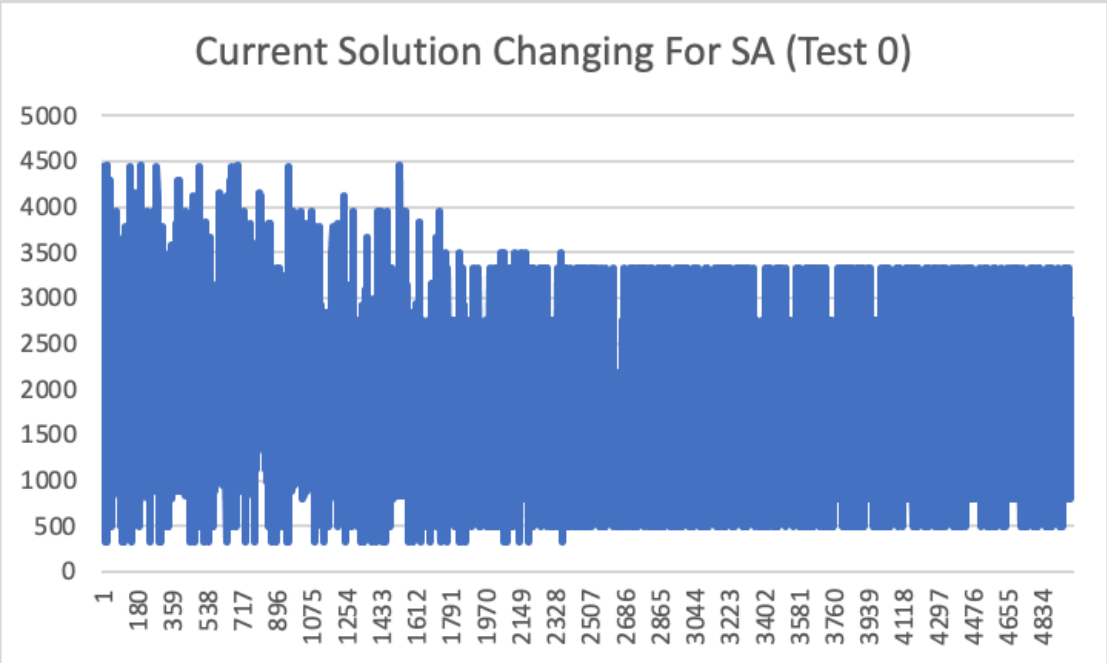
Cooling Coefficie	# of wrong result
0.6	1
0.7	2
0.8	0
0.9	0
0.99	0

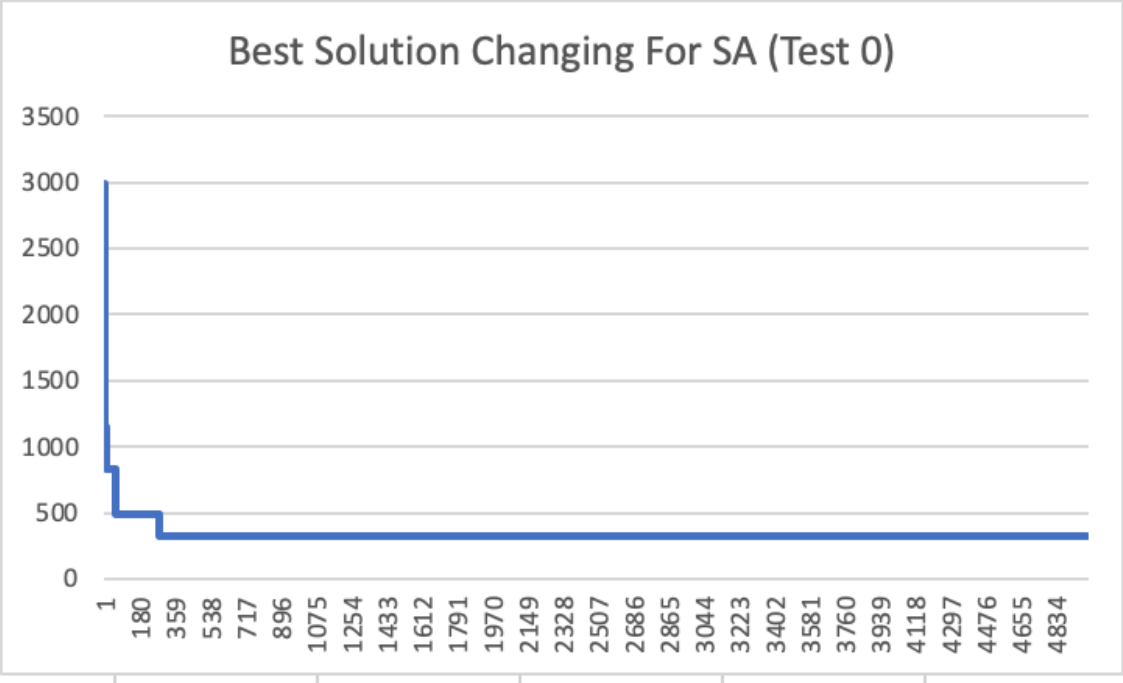
Again, as seen in the table, the best results were obtained at 0.8-0.9 and 0.99 values. When I look at the temperature changes for these three values, I saw that it concentrates too much on 0s with a coefficient of 0.8 and that it is very high at 0.99, so I chose the 0.9 coefficient as a final to obtain the most homogeneous amount of circulation.

Running For [0.9 cooling coefficient, 100 outer loop size, 50 inner loop size and total waste as an initial temperature]



Since it is necessary to wait approximately 30 minutes for the above result(all of above are accurate result) to appear, I added the part of the current code that will only work until the 5th test in the main function. Apart from that, in skipped tests, the program freezed for a reason I could not identify, so I skipped them by pressing “\n” in the running view.


Current Solution Changing For SA (Test 0)
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
1
180
359
538
717
896
1075
1254
1433
1612
1791
1970
2149
2328
2507
2686
2865
3044
3223
3402
3581
3760
3939
4118
4297
4476
4655
4834


Best Solution Changing For SA (Test 0)
3500
3000
2500
2000
1500
1000
500
0
1
180
359
538
717
896
1075
1254
1433
1612
1791
1970
2149
2328
2507
2686
2865
3044
3223
3402
3581
3760
3939
4118
4297
4476
4655
4834

Tabu Search Algorithm (HW5)

For the Tabu Search solution, I applied the following target frame:

Algorithm (Local improvement process)

1. Set tabu list to null ,set x_o;
2. **repeat**
3. $\{x_1, x_2, ..., x_n\}$ =Generate_neighborhood (x_o);
4. **for** i =1 *to* n
5. **if** tabu(i) is 0;
6. **if** x_i meet aspiration criterion;
7. replace x_b by x_i, $x_o = x_i$ and update tabu list;
8. **else if** x_i is the best solution in $\{x_1, x_2, ..., x_i\}$
9. $x_o = x_i$, update tabu list;
10. **end**
11. **end**
12. **end**
13. **until** a termination condition is satisfied
14. Return x_b as the best individual.

I used the tabu list, tabu tenure, aspiration criteria and termination criteria approaches, which are the main approaches in the tabu search algorithm, but I did not use long term memory. The reason for this is that it was a program with optimal results in general without using it. I did not want to slow down the algorithm, which is already a bit slow, because of the difficulty of implementation and the possibility of computational complexity.

In this warehouse problem, I think it like Orienteering Problem. So I represent the this warehouse problem like a permutation problem. Although this problem is not permutation problem,I changed the chosen case by changing case selecting order with neigbourhood structure in candidate list. Also I define the feasibility limit for changing and inserting line.

After the neighborhood definition. For the Initial solution, it was necessary to create a permutation with the solution, I simply did random permutation creation for problems. I decided the adequacy according to the smalling of the increasing amount as in the objective function.

Choices Made For The Main Approaches

Solution representation for this problem:

Orders		
	Product 1	Product 2
Order 1	2	0
Order 2	2	4
Order 3	2	0
Order Total	6	4

Cases		
	Product 1	Product 2
Case 1	10	0
Case 2	0	21
Case 3	6	3
Case 4	8	4

As the permutation problem these are the solutions:

- 1, 2, 3, 4 → Initial solution for this problem.

- 4 | 2, 3, 1 → The feasibility limit is 1 in this permutation case.
- 1, 3, 2 | 4 → The feasibility limit is 3 in this permutation case.
- 3, 4 | 1, 2 → The feasibility limit is 2 in this permutation case.
- .
- .
- .
- .
-
- 4! permutations

Cost of a solution:

Simply summing up the cases from the beginning of the permutation to the feasible limit on a product-by-product basis. It is then obtained by subtracting and adding all of the total product values in the order.

1, 3, 2 | 4 → [10,0]+[6,3]+[0,21] = [16,24]
[16,24] – [6,4] = [10,20] so there are 10 waste in P1 and 20 waste in P2
At the end total waste or cost is 10+20=30

Initial solution generation:

First of all, the indexes of each case are written into the permutation, and then each index is changed with another random index and the permutation is traversed and changed.

1, 2, 3, 4 →Random Exchange Loop→ 2, 4, 3, 1

Neighbourhood:

Exchange Neighborhood: Finds randomly both different index1 and index2 in the permutation . Then it exchange the value in the index1 and value in the index2.

1, 3, 2, 4 → 1, 4, 2, 3

Tabu List:

First, I created a list with 0 as the number of cases. Then it updates the 0 in the selected case index in each loop to 7 as a tabu tenure. It then decrements by 1 for each value greater than 0 in each TS loop.

Tabu Tenure:

I chose 7(magic number) as taboo tenure. In this TS algorithm I use static tabu tenure strategy.

Aspiration Rule:

There are two aspiration criteria. First, if a result that develops the best result found in the TS loop in the candidate list is found without any tabus. Second, if there is no non-tabu solution in the candidate list, it chooses the best locally in the candidate list as the current solution.

Long-term Memory:

I didn't use long term memory

Termination Criteria:

I have specified the maximum number of loops as termination criteria. This number of loops is 100.

Pseudocodes of the TS algorithm

```
void TS()
     termination_condition=100
     candidate_size=50
     tabu_tenure=7

     vector<int> initial_solution
     initial_solution=create_random_initial_solution(…)

     vector<int> tabu_list
     fill tabu_list by number of cases "0" in problem

     vector<int> best_solution
     vector<vector<int> > candidate_list

     Repeat the following sequence until the termination_cond.
          candidate_list=create_candidate_list(..)
          vector<int> local_best
          vector<int> non_tabu_best

          for i in candidate list
               find fitness value of candidate_list[i] by
               calculating objective function

               if candidate_list[i] better than local_best
                    local_best=candidate_list[i]
               end if
               if candidate_list[i] is not tabu
                    if candidate_list[i] better than
                    non_tabu_best
                         non_tabu_best=candidate_list[i]
                    end if
               end if
          end for

          decrease_tabu_list(..)

          if local_best better than best_solution
               best_solution=local_best
               initial_solution=local_best
               add tabu tenures
          else if there is non tabu solution
               initial_solution=non_tabu_best
               add tabu tenures
          else
               initial_solution=local_best
               add tabu tenures
```

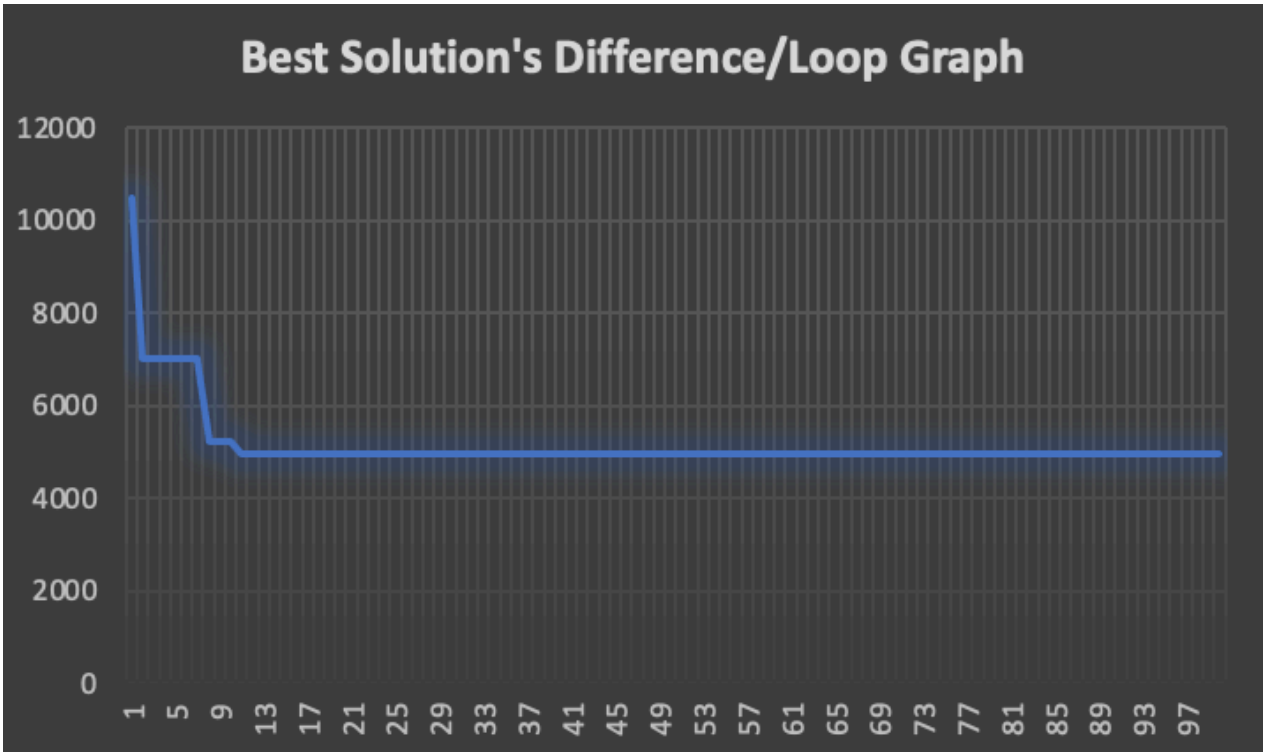
```
            end if

      end repeat

      print best_solution

end function


vector<vector<int> > create_candidate_list(vector<int>
solution,int size,vector<vector<int> >& changed_value_list)

      vector<vector<int> > candidate_list

      for i=0 to i==size
            vector<int> changed_values
            candidate_list.push_back(
            exchange_neighbourhood(solution,ch anged_values) )

            changed_value_list.push_back(changed_values)
      end for

      return candidate_list

end function
```

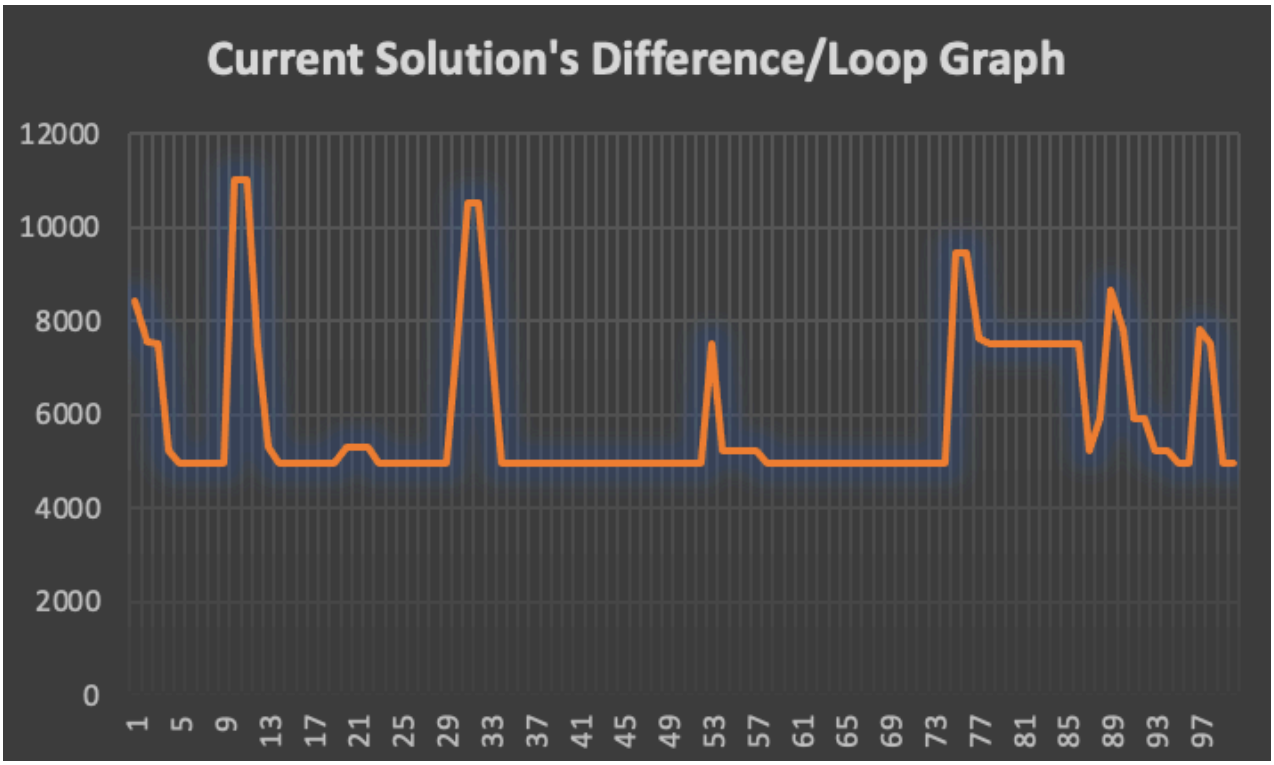
- **Tests And Hyper Parameters Given Text File (HW5)**

I used 3 hyper parameters in this algorithm, these are:

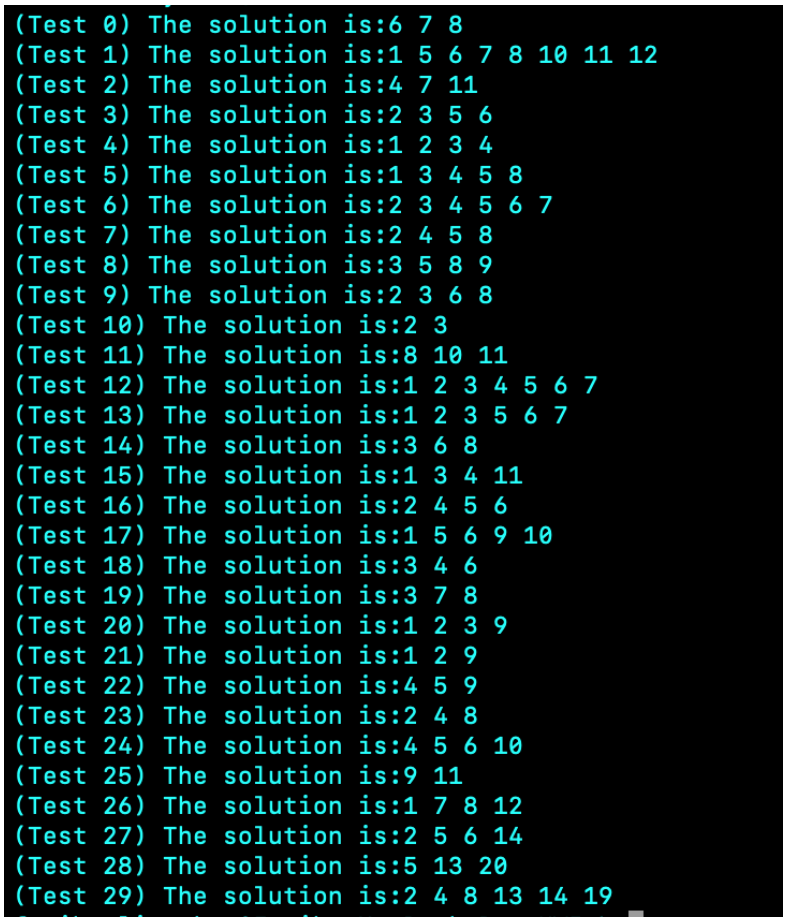
Termination condition=100
Candidate size=50
Tabu tenure=7

Changes of best solutions and current solutions in TS loop (for test1)





Running for first 30 tests in test data



26 best results achieved in 30 test data

Genetic Algorithm (HW6)

For the Genetic Algorithm solution, I applied the following target frame:

Algorithm 1 The pseudocode of a GA

```
Set parameters
Choose encode method
Generate the initial population
while i < MaxIteration and Bestfitness < MaxFitness do
    Fitness calculation
    Selection
    Crossover
    Mutation
end while
Decode the individual with maximum fitness
return the best solution
```

In this warehouse problem, I think it like Orienteering Problem. So I represent the this warehouse problem like a permutation problem. Although this problem is not permutation problem,I changed the chosen case by changing case selecting order with neigbourhood structure in candidate list. Also I define the feasibility limit for printing.

For this problem, I used a method similar to the simple genetic algorithm. Similarly, I completely changed the parent population to the child population in survivor selection. As a difference, I didn't use the strict requirement of binary representation. Instead, I thought the phenotype and genotype space the same and placed the indexes of the cases here. The reason for this is that I treated the problem like a permutation problem, as I mentioned. In addition, due to the way the problem is handled, the mutation and crossover part was handled differently from SGA.

For the Initial solution population, it was necessary to create a permutation with the solution, I simply did random permutation creation for problems and add the

population list. I decided the adequacy according to the smalling of the increasing amount as in the objective function.

Choices Made For The Main Approaches

Solution representation for this problem:

Orders		
	Product 1	Product 2
Order 1	2	0
Order 2	2	4
Order 3	2	0
Order Total	6	4

Cases		
	Product 1	Product 2
Case 1	10	0
Case 2	0	21
Case 3	6	3
Case 4	8	4

As the permutation problem these are the solutions:

- 1, 2, 3, 4 → Initial solution for this problem.

- 4 I 2, 3, 1 → The feasibility limit is 1 in this permutation case.
- 1, 3, 2 I 4 → The feasibility limit is 3 in this permutation case.
- 3, 4 I 1, 2 → The feasibility limit is 2 in this permutation case.
- .
- .
- .
- 4! Permutations

Cost of a solution:

Simply summing up the cases from the beginning of the permutation to the feasible limit on a product-by-product basis. It is then obtained by subtracting and adding all of the total product values in the order.

1, 3, 2 I 4 → [10,0]+[6,3]+[0,21] = [16,24]
[16,24] – [6,4] = [10,20] so there are 10 waste in P1 and 20 waste in P2
At the end total waste or cost is 10+20=30

Initial solution generation:

First of all, the indexes of each case are written into the permutation, and then each index is changed with another random index and the permutation is traversed and changed.

1, 2, 3, 4 →Random Exchange Loop→ 2, 4, 3, 1

Population Generation:

The chromosome (case permutation) was created and added together with the method used to create the initial solution in random order as much as the population size considered as a hyper parameter.

1, 2, 3, 4 →Random Exchange Loop→ 2, 4, 3, 1
1, 2, 3, 4 →Random Exchange Loop→ 3, 1, 4, 2
1, 2, 3, 4 →Random Exchange Loop→ 1, 3, 4, 2
1, 2, 3, 4 →Random Exchange Loop→ 4, 2, 3, 1

Mating pool/population selection:

The roulette wheel selection system was used to create a good mating pool from the randomly initialized or ancestral population. That is, if the objective function was the smallest, a random selection was made by dividing it into numerators in a way to provide the greatest probability of being selected.

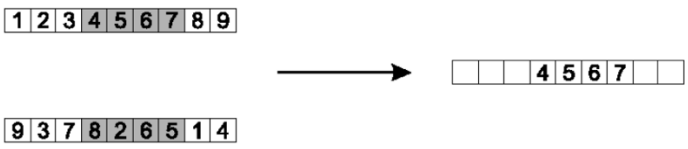
Survivor selection:

I used the age-based selection strategy with full changing the old population.

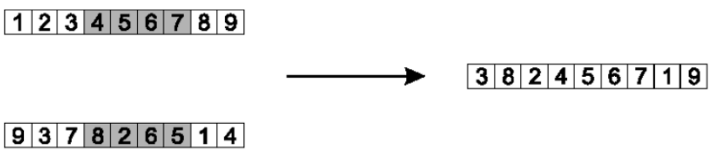
Crossover:

In the crossover part, an order 1 crossover was used with 0.8 probability. It worked like this to create 2 children using 2 parents:

- Copy randomly selected set from first parent



- Copy rest from second parent in order 1,9,3,8,2



Mutation:

A random gene is determined to occur 1 time in the whole chromosome and replaced with another randomly determined gene.

Exchange Neighborhood: Finds randomly both different index1 and index2 in the permutation . Then it exchange the value in the index1 and value in the index2.

1, 3, 2, 4 → 1, 4, 2, 3

Termination Criteria:

I have specified the maximum number of loops as termination criteria. This number of loops is 100.

Pseudocodes of the GA algorithm

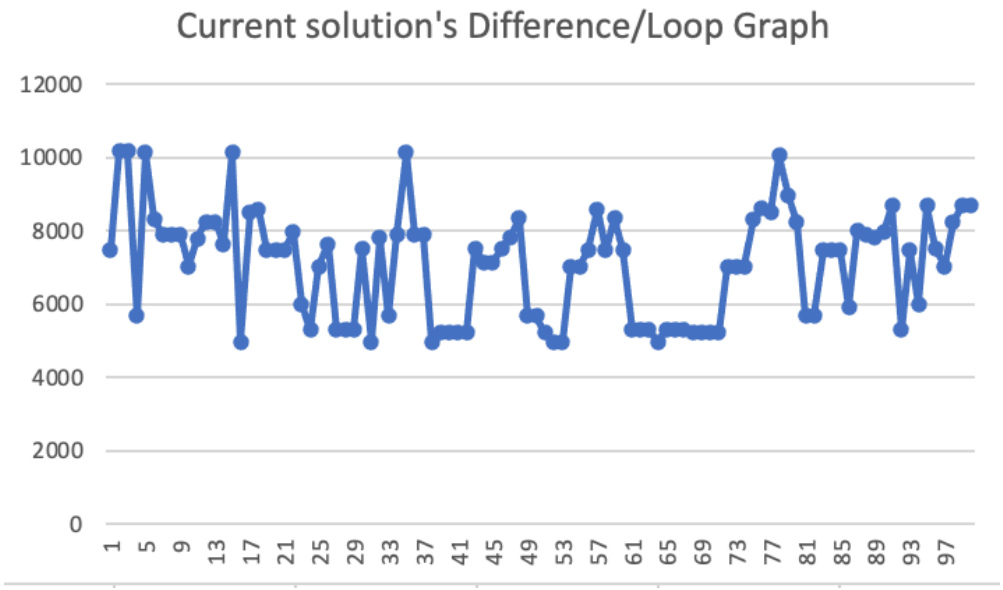
```
void GA()
     termination_criteria=100
     population_size=20

     vector<vector<int> >
     population=random_population_generate(population_size)

     bests_difference=MAX_INT
     vector<int> solution
     for i=0 to i==termination_criteria
          vector<vector<int> >
          mating_pool=create_mating_pool(population)

          O1_crossover(mating_pool)
          mutation(mating_pool)

          initialize local_best_difference
          vector<int>
          local_best=find_best_solution(local_best_difference)

          if local_best_difference<bests_difference
               bests_difference=local_best_difference
               solution=local_best
          end if

          population=mating_pool
     end for
     print solution
end function
```

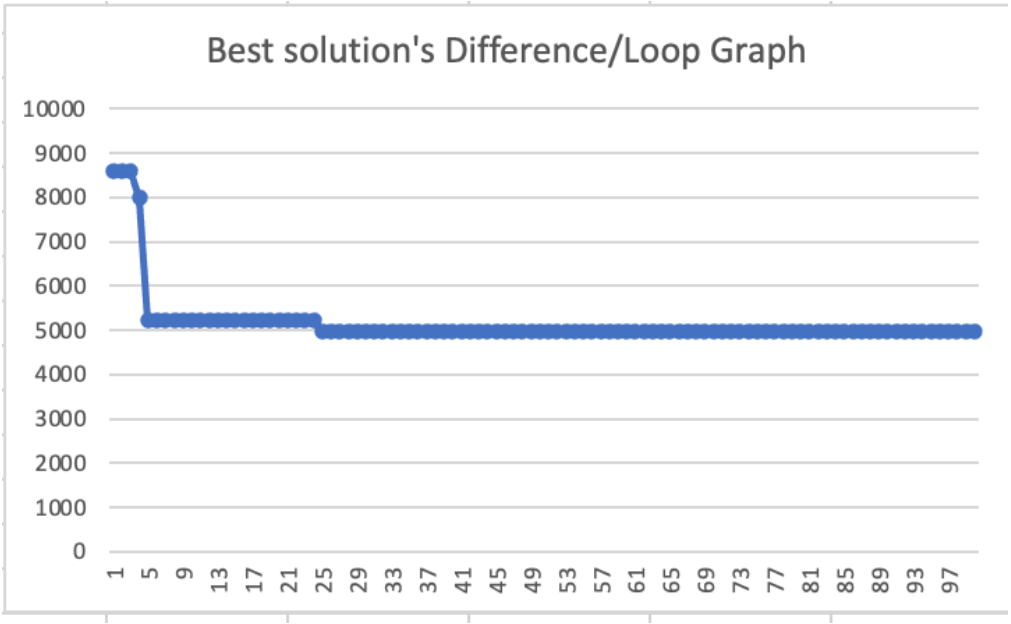
Hyper Parameters

I used 4 hyper parameters in this algorithm, these are:

Termination condition=100
Population size=20
Crossover probability=0.8
Mutation prorability=1/chromosome length(In this case: 20)

Changes of best solutions and current solutions in GA loop (for test1)





- **Tests And Hyper Parameters Given Text File (HW6)**

Running for first 24 tests in test data (termination criteria=100,population size=20)

```
fatihselimyakar@Fatihs-MacBook-Pro HW6 % ./run
(Test 0) The solution is:6 7 8
(Test 1) The solution is:1 5 6 7 8 10 11 12
(Test 2) The solution is:4 7 11
(Test 3) The solution is:2 3 5 6
(Test 4) The solution is:1 2 3 4
(Test 5) The solution is:1 3 4 5 8
(Test 6) The solution is:2 3 4 5 6 7
(Test 7) The solution is:2 4 5 8
(Test 8) The solution is:3 5 8 9
(Test 9) The solution is:2 3 6 8
(Test 10) The solution is:2 3
(Test 11) The solution is:8 10 11
(Test 12) The solution is:1 2 3 4 5 6 7
(Test 13) The solution is:1 2 3 5 6 7
(Test 14) The solution is:3 6 8
(Test 15) The solution is:1 3 4 11
(Test 16) The solution is:2 4 5 6
(Test 17) The solution is:1 5 6 9 10
(Test 18) The solution is:1 3 8 10
(Test 19) The solution is:2 7 10
(Test 20) The solution is:1 2 3 9
(Test 21) The solution is:1 2 9
(Test 22) The solution is:4 5 9
(Test 23) The solution is:2 4 8
```

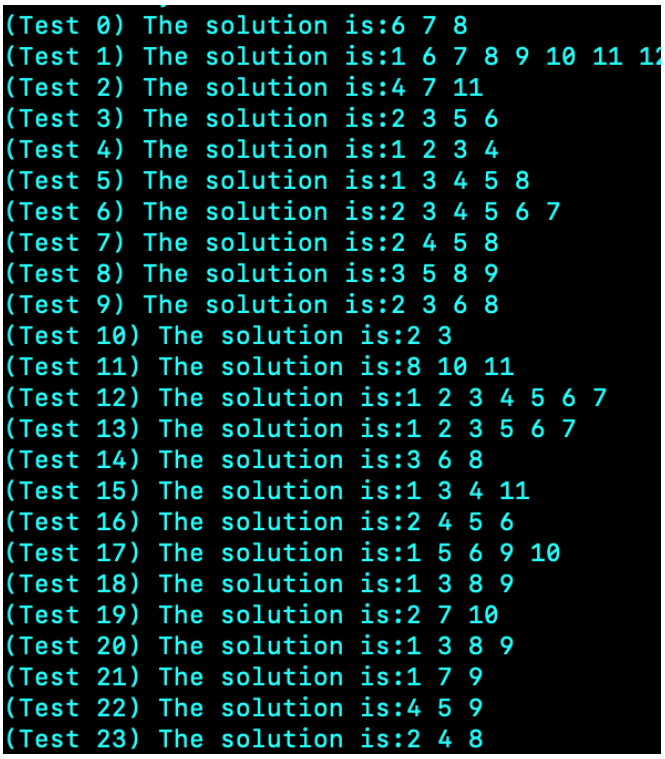
It found 24/24 optimal result. In submission, this combination used. It can take long time while running because of this combination.

Running for first 24 tests in test data (termination criteria=50,population size=20)

```
(Test 0) The solution is:6 7 8
(Test 1) The solution is:1 5 6 7 8 10 11 12
(Test 2) The solution is:4 7 11
(Test 3) The solution is:2 3 5 6
(Test 4) The solution is:1 2 3 4
(Test 5) The solution is:1 3 4 5 8
(Test 6) The solution is:2 3 4 5 6 7
(Test 7) The solution is:2 4 5 8
(Test 8) The solution is:3 5 8 9
(Test 9) The solution is:2 3 6 8
(Test 10) The solution is:2 3
(Test 11) The solution is:8 10 11
(Test 12) The solution is:1 2 3 4 5 6 7
(Test 13) The solution is:1 2 3 5 6 7
(Test 14) The solution is:3 6 8
(Test 15) The solution is:1 3 4 11
(Test 16) The solution is:2 4 5 6
(Test 17) The solution is:1 5 6 9 10
(Test 18) The solution is:1 3 8 10
(Test 19) The solution is:2 7 10
(Test 20) The solution is:1 2 3 9
(Test 21) The solution is:1 2 9
(Test 22) The solution is:4 5 9
(Test 23) The solution is:2 4 8
```

It found 24/24 optimal result.

Running for first 23 tests in test data (termination criteria=25,population size=20)



It found 20/24 optimal result.

Ant Colony Algorithm (HW7)

For the Ant Colony solution, I applied the following target frame:

For $t = 1, \ldots, t_{max}$
  For each ant $k = 1, \ldots, m$
    Choose a city randomly
    For each non visited city i
      Choose a city j, from the list J_i^k of remaining cities, according to

$$p_{ij}^k(t) = \begin{cases} \frac{(\tau_{ij}(t))^\alpha \cdot (\eta_{ij})^\beta}{\sum_{l \in J_i^k} (\tau_{il}(t))^\alpha \cdot (\eta_{ij})^\beta} & \text{if } j \in J_i^k \\ 0 & \text{if } j \notin J_i^k \end{cases}$$

    End For
    Deposit a trail $\Delta\tau_{ij}^k(t)$ on the path $T^k(t)$ in accordance with

$$\Delta\tau_{ij}^k(t) = \begin{cases} \frac{Q}{L^k(t)} & \text{if } (i,j) \in T^k(t) \\ 0 & \text{if } (i,j) \notin T^k(t) \end{cases}$$

  End For
  Evaporate trails according to

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \Delta\tau_{ij}(t)$$

End For

In this warehouse problem, I think it like Orienteering Problem. So I represent the this warehouse problem like a permutation problem. Although this problem is not permutation problem,I changed the chosen case by changing case selecting order with neigbourhood structure in candidate list. Also I define the feasibility limit for printing.

Of course ant colony system was not applied to this problem as applied to tsp. I collected some data to make it applicable and applied it accordingly.

For the Initial solution I use the basic greedy solution path and initialize the pheromones by greedy solution path and quality.

Choices Made For The Main Approaches

Solution representation for this problem:

Orders		
	Product 1	Product 2
Order 1	2	0
Order 2	2	4
Order 3	2	0
Order Total	6	4

Cases		
	Product 1	Product 2
Case 1	10	0
Case 2	0	21
Case 3	6	3
Case 4	8	4

As the permutation problem these are the solutions:

- 1, 2, 3, 4 → Initial solution for this problem.

- 4 | 2, 3, 1 → The feasibility limit is 1 in this permutation case.
- 1, 3, 2 | 4 → The feasibility limit is 3 in this permutation case.
- 3, 4 | 1, 2 → The feasibility limit is 2 in this permutation case.
- .
- .
- .
- 4! Permutations

Cost of a solution:

Simply summing up the cases from the beginning of the permutation to the feasible limit on a product-by-product basis. It is then obtained by subtracting and adding all of the total product values in the order.

1, 3, 2 | 4 → [10,0]+[6,3]+[0,21] = [16,24]
[16,24] – [6,4] = [10,20] so there are 10 waste in P1 and 20 waste in P2
At the end total waste or cost is 10+20=30

Initial solution generation:

I created the initial solution and chose a greedy solution to secrete pheromone accordingly and started it over this solution. I determined the amount of pheromone here as 1 / solution cost (difference). If we accept the following sequence as a greedy solution, pheremons are distributed to the matrix as follows:

1, 2, 3, 4 →

PH			
	PH		
		PH	

City Structure:

The cases in our problem were used to establish the city structure in TSP. So each case was considered a city. As for the distances between cities, for the distances of cities in index 0 and index 1, the sum of cases in index 0 and index 1 is subtracted from the limit value and the absolute value is taken, and then for the n reverse is taken according to multiplication. To formulate:

$$Distance(0,1) = |(cases[0] + cases[1]) - order\ sums)|$$

$$n = 1/distance(0,1)$$

Pheromone Deposition:

To determine the amount of pheromone, a method similar to distance calculation was used. Here, how low the distance of the whole path of the ant from the target value was was determined as quality. In other words, the amount of pheromone for the p solution was determined as follows:

$$\Delta T_{ij} = 1/path_{quality}$$
$$path_{quality} = path's\ case\ sum - order'sum$$

New City Selection:

To select a new city, roulette wheel selection was made among the possibilities of unselected cities determined by the formula below:

$$p_{ij}^{k}(t) = \begin{cases} \frac{(\tau_{ij}(t))^{\alpha} \cdot (\eta_{ij})^{\beta}}{\sum_{l \in J_i^k} (\tau_{il}(t))^{\alpha} \cdot (\eta_{ij})^{\beta}} & \text{if } j \in J_i^k \\ 0 & \text{if } j \notin J_i^k \end{cases}$$

Trail Evoporation:

The following formula was used to evoporate:

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \Delta\tau_{ij}(t)$$

Termination Criteria:

I have specified the maximum number of loops as termination criteria. This number of loops is 10.

Pseudocodes of the ACO algorithm

```
void ACO()
     ant_size=case_size

     vector<int> initial_solution=greedy_algorithm()
     difference = find_feasible_limit(initial_solution)
     initializePheromones(1.0/difference,initial_solution)

     best_solution=MAX_INT
     best_feasible_limit=0;
     vector<int> best_sol_vec
     initialize vector<vector<float> > pheromone_matrix_local
     for i=0 to i==termination_criteria
          fill pheremone_matrix_local
          for ant=0 to ant==ant_size
               create cities<int>,ant_path<int> vectors
               index=random city index
               city=cities[index]
               while(1)
                    add city in ant_path
                    delete index'ed city in cities

                    if cities size equal to 0
                         break
                    end if

                    probs=remaining_city_probs(cities)

                    select a new city using roulette Wheel
                    selection

                    index=new index
                    city=new city

               end while
               if current solution better than best
                    update bests
```

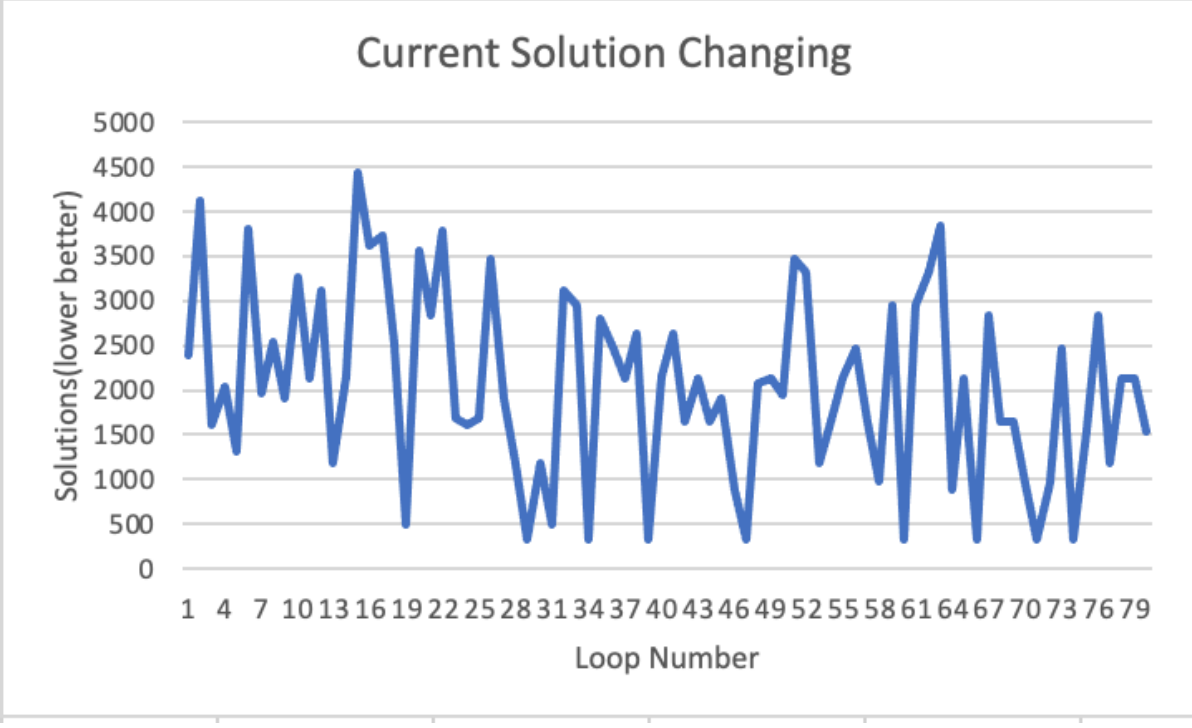
```
                end if
                Deposit a trail
            end for
            Evaporate trails
        End for
        print solution
end function
```

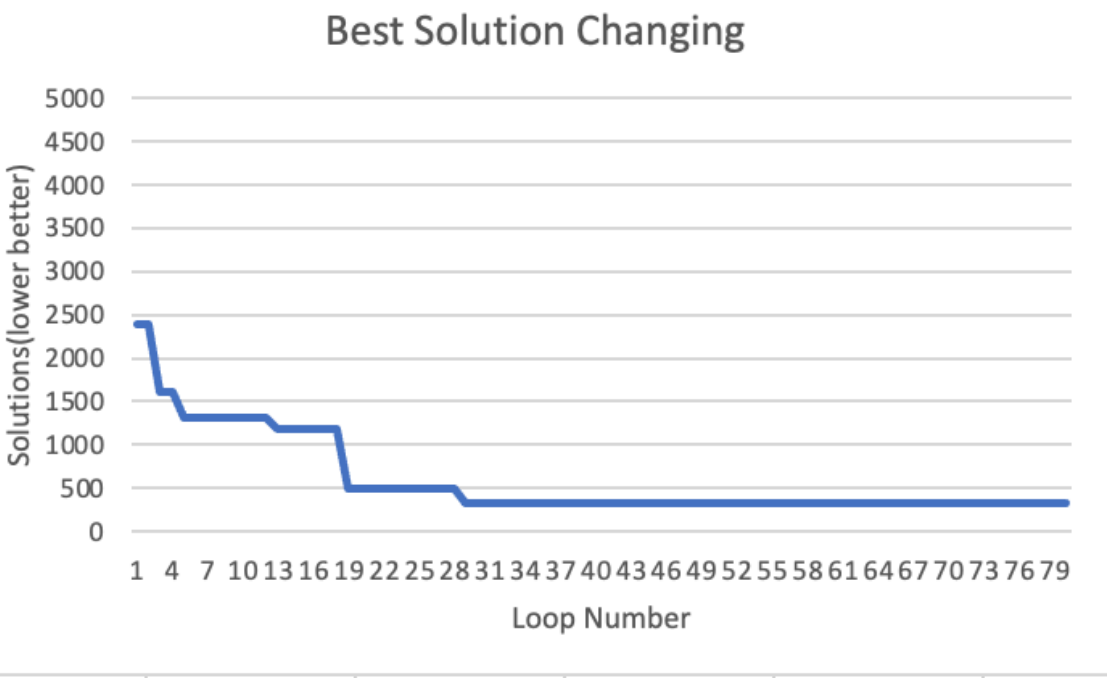
Hyper Parameters

I used 4 hyper parameters in this algorithm, these are:

Termination condition=10
Population size=case size
p=0.3
Alpha=2
Beta=1

Changes of best solutions and current solutions in ACO loop (Test 0)





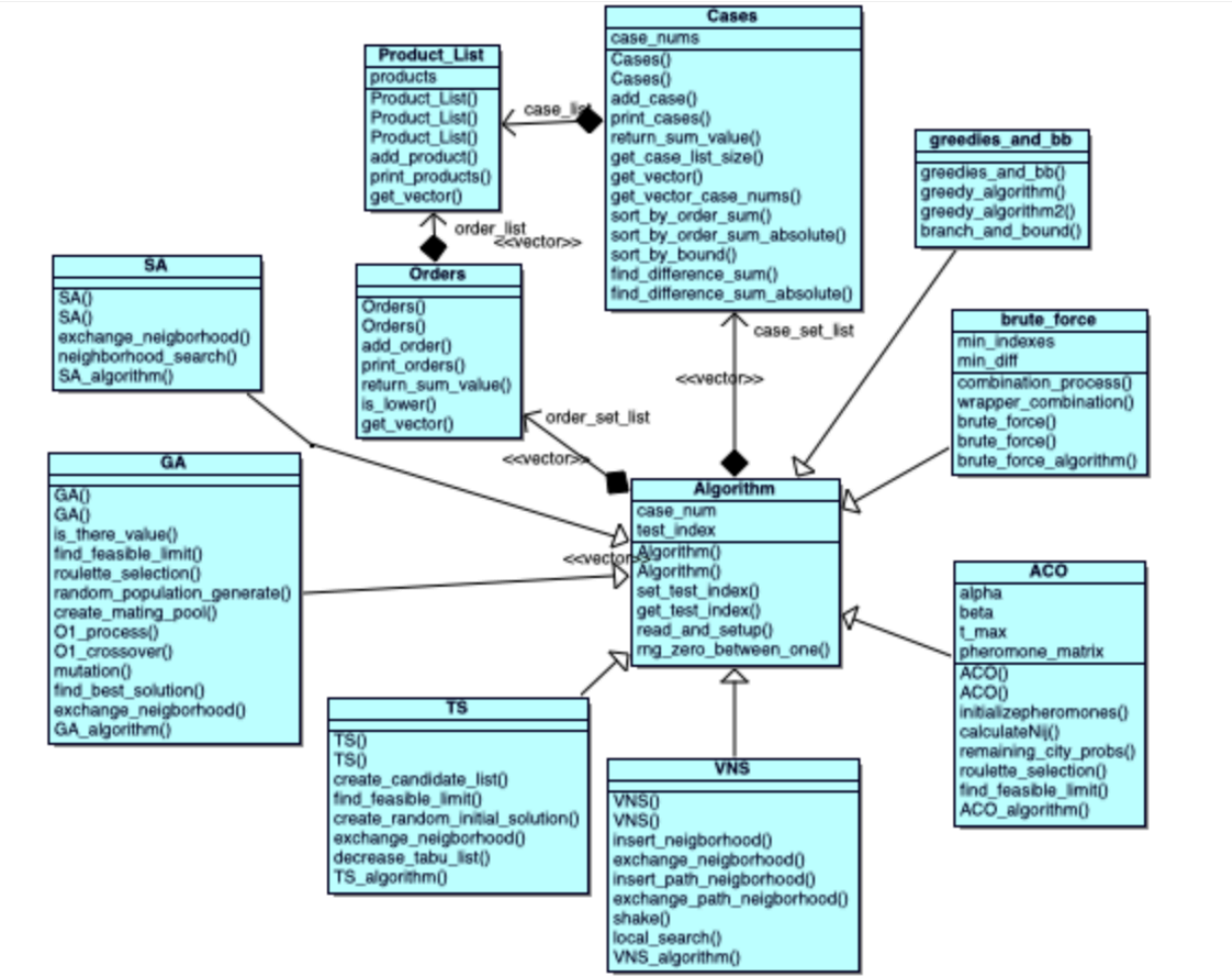
- **Tests Parameters Given Text File (HW7)**

Running for first 24 tests in test data

```
(Test 0) The solution is:7 8 6
(Test 1) The solution is:8 11 9 12 7 1 6 10
(Test 2) The solution is:3 1 2 7
(Test 3) The solution is:5 6 2 3
(Test 4) The solution is:3 1 4 2
(Test 5) The solution is:5 4 8 3 6
(Test 6) The solution is:3 6 5 7 4 2
(Test 7) The solution is:2 5 8 4
(Test 8) The solution is:9 3 5 8
(Test 9) The solution is:3 8 2 6
(Test 10) The solution is:2 3
(Test 11) The solution is:8 11 10
(Test 12) The solution is:6 5 3 7 4 1 2
(Test 13) The solution is:2 5 1 3 4 7 6
(Test 14) The solution is:3 6 8
(Test 15) The solution is:1 11 4 3
(Test 16) The solution is:5 6 4 2
(Test 17) The solution is:6 10 9 5 1
(Test 18) The solution is:5 1 9
(Test 19) The solution is:7 8 3
(Test 20) The solution is:8 1 9 3
(Test 21) The solution is:1 2 9
(Test 22) The solution is:1 7 9 8
(Test 23) The solution is:8 2 4
```

It found 14/24 optimal result. It can take long time while running.

General Implementation's UML Diagram



General Optimal Solutions

Algorithms	Brute For	Greedy1	Greedy2	BB	VNS	SA	TS	GA	ACO
# Of Optimal Solution	24/24	0/24	0/24	0/24	24/24	24/24	20/24	24/24	14/24