name: Yun Zhou, ID: 300442776

Part 1:

I am using the **Google OR-Tools** for this part, the results for each dataset can be observed from below screenshots clearly.

The small data set results:

```
print('-'*80)
print("Small Cloud Resource Allocation dataset:")
print('-'*20)
smallDS.define_maths_models()
smallDS.solve_assignment_problem()
print('-'*80)
print()
```

```
--------------------------------------------------------------------------------
Small Cloud Resource Allocation dataset:
--------------------
Objective value(Total Payment) =5647.0
Total CPU used = 886.0 out of the CPU Capacity:1000
Total Memory used = 1989.0 out of the Memory Capacity:2000
Solution: (The obtained  xi values):
 Value is 1, which means they are selected:
        ['x[0]', 'x[1]', 'x[2]', 'x[4]', 'x[5]', 'x[6]', 'x[7]', 'x[8]', 'x[9]']
All:
                x[0] : 1.0
        x[1] : 1.0
        x[2] : 1.0
        x[3] : 0.0
        x[4] : 1.0
        x[5] : 1.0
        x[6] : 1.0
        x[7] : 1.0
        x[8] : 1.0
        x[9] : 1.0

---------------
Statistics
 Problem solved in 5.000000 milliseconds
 Problem solved in 0 iterations
 Problem solved in 0 branch-and-bound nodes
--------------------------------------------------------------------------------
```

The large data set results :

```
print('-'*80)
print("Large Cloud Resource Allocation dataset:")
print('-'*20)
largeDS.define_maths_models()
largeDS.solve_assignment_problem()
print('-'*80)
```

```
--------------------------------------------------------------------------------
Large Cloud Resource Allocation dataset:
--------------------
Objective value(Total Payment) =29459.0
Total CPU used = 9997.0 out of the CPU Capacity:10000
Total Memory used = 9859.0 out of the Memory Capacity:10000
Solution: (The obtained  xi values):
 Value is 1, which means they are selected:
        ['x[4]', 'x[34]', 'x[71]', 'x[147]', 'x[162]', 'x[176]', 'x[178]', 'x[199]', 'x[244]', 'x[284]',
'x[320]', 'x[326]', 'x[346]', 'x[400]', 'x[410]', 'x[466]', 'x[508]', 'x[511]', 'x[539]', 'x[625]', 'x[67
9]', 'x[680]', 'x[712]', 'x[751]', 'x[759]', 'x[767]', 'x[862]', 'x[866]', 'x[897]', 'x[915]', 'x[919]',
'x[941]', 'x[969]', 'x[982]', 'x[985]']
All:
                x[0] : 0.0
        x[1] : 0.0
        x[2] : 0.0
        x[3] : 0.0
        x[4] : 1.0
        x[5] : 0.0
        x[6] : 0.0
        x[7] : 0.0
        x[8] : 0.0
        x[9] : 0.0
        x[10] : 0.0
        x[11] : 0.0
        x[12] : 0.0
        x[13] : 0.0
        x[14] : 0.0
        x[15] : 0.0
        x[16] : 0.0
        x[17] : 0.0
        x[18] : 0.0
        x[19] : 0.0
        x[20] : 0.0
        x[21] : 0.0
        x[22] : 0.0
        x[23] : 0.0
```

```
x[984] : 0.0
x[985] : 1.0
x[986] : 0.0
x[987] : 0.0
x[988] : 0.0
x[989] : 0.0
x[990] : 0.0
x[991] : 0.0
x[992] : 0.0
x[993] : 0.0
x[994] : 0.0
x[995] : 0.0
x[996] : 0.0
x[997] : 0.0
x[998] : 0.0
x[999] : 0.0

---------------
Statistics
 Problem solved in 250.000000 milliseconds
 Problem solved in 2011 iterations
 Problem solved in 290 branch-and-bound nodes
-----------------------------------------------
```

1. The solution (the obtained x_i^* values)

- For the small dataset, the obtained x_i^* values are:

```
Total Memory used = 1989.0 out of the Memory Capacity:2000
Solution: (The obtained  xi values):
 Value is 1, which means they are selected:
        ['x[0]', 'x[1]', 'x[2]', 'x[4]', 'x[5]', 'x[6]', 'x[7]', 'x[8]', 'x[9]']
All:
                x[0] : 1.0
        x[1] : 1.0
        x[2] : 1.0
        x[3] : 0.0
        x[4] : 1.0
        x[5] : 1.0
        x[6] : 1.0
        x[7] : 1.0
        x[8] : 1.0
        x[9] : 1.0

---------------
Statistics
 Problem solved in 5.000000 milliseconds
 Problem solved in 0 iterations
 Problem solved in 0 branch-and-bound nodes
-------------------------------------------------------------------------------
```

- For the large dataset, the obtained x_i^* values are displayed below. It is clearly to see which x_i has the value of 1.

```
Solution: (The obtained  xi values):
 Value is 1, which means they are selected:
        ['x[4]', 'x[34]', 'x[71]', 'x[147]', 'x[162]', 'x[176]', 'x[178]', 'x[199]', 'x[244]', 'x[284]',
'x[320]', 'x[326]', 'x[346]', 'x[400]', 'x[410]', 'x[466]', 'x[508]', 'x[511]', 'x[539]', 'x[625]', 'x[67
9]', 'x[680]', 'x[712]', 'x[751]', 'x[759]', 'x[767]', 'x[862]', 'x[866]', 'x[897]', 'x[915]', 'x[919]',
'x[941]', 'x[969]', 'x[982]', 'x[985]']
All:
                x[0] : 0.0
        x[1] : 0.0
        x[2] : 0.0
        x[3] : 0.0
        x[4] : 1.0
        x[5] : 0.0
        x[6] : 0.0
        x[7] : 0.0
        x[8] : 0.0
        x[9] : 0.0
        x[10] : 0.0
```

2. The objective value, the total CPU used and total memory used for each dataset

- For the small dataset,

```
--------------------------------------------------------------------------------
Small Cloud Resource Allocation dataset:
--------------------
Objective value(Total Payment) =5647.0
Total CPU used = 886.0 out of the CPU Capacity:1000
Total Memory used = 1989.0 out of the Memory Capacity:2000
```

- For the large dataset :

```
--------------------------------------------------------------------------------
Large Cloud Resource Allocation dataset:
```

```
--------------------
Objective value(Total Payment) =29459.0
Total CPU used = 9997.0 out of the CPU Capacity:10000
Total Memory used = 9859.0 out of the Memory Capacity:10000
```

In addition , some important corresponding code that **define the integer decision variable x_i, define the objective function and define the 2 constraints** are displayed below : They are obtained from my source code.

PYTHON

```python
# define variable, we only need N number of x for each job
self.x = {}
for i in range(self.N):
    self.x[i] = self.solver.IntVar(0, 1, 'x[%i]' % i)

# assign the constraints and objective function
constraint_expr1,constraint_expr2, obj_express= [],[],[]
for i in range(self.N):
    constraint_expr1.append(self.jobs['CPUDemand'][i] * self.x[i])
    constraint_expr2.append(self.jobs['MemoryDemand'][i] * self.x[i])
    obj_express.append(self.jobs['payment'][i] * self.x[i])
# define constraints
self.solver.Add(sum(constraint_expr1)<= self.Q1,"cpu_capacity_constraint")
self.solver.Add(sum(constraint_expr2) <= self.Q2,"memory_capacity_constraint")
# define objective function
self.solver.Maximize(sum(obj_express))
```

The result for total CPU and memory used are computed manually which can be observed from the below code.

PYTHON

```python
total_cpu_used , total_memory_used = 0,0
x_sol_vals_str = ""
for j in range(self.N):
    x_temp = self.x[j].solution_value()
    x_sol_vals_str += f"\t{self.x[j].name()} : {x_temp} \n"
    # count the total cpu and memory used
    total_cpu_used += self.jobs['CPUDemand'][j] * x_temp
    total_memory_used += self.jobs['MemoryDemand'][j] * x_temp
print(f'Objective value(Total Payment) ={self.solver.Objective().Value()}')
print(f"Total CPU used = {total_cpu_used} out of the CPU Capacity:{self.Q1}")
print(f"Total Memory used = {total_memory_used} out of the Memory Capacity:{self.Q2}")
print(f"Solution: (The obtained  xi value): \n{x_sol_vals_str}")
```

Part 2: Greedy Heuristic

1. clearly describe the two heuristics, i.e., sorting criteria of the jobs you designed.

My first greedy heuristic is to **sort the jobs in the decreasing order of their payment/CPUDemand ratio,** and then selects the sorted jobs one by one until we cannot add more jobs. The stopping criteria refer to the scenario that either CPU or memory demand of any remaining job will exceed total capacity.

PYTHON

```python
# define the efficiency by adding payment per cpuDemand and payment per memoryDemand
efficiency = [ds.get_jobs().iloc[i]['payment'] / ds.get_jobs().iloc[i]['CPUDemand']
              for i in range(len(ds.get_jobs()))]
# add the efficiency to the df
ds.get_jobs()['sorting criterion'] = efficiency
```

My second greedy heuristic is to **sort the jobs in the decreasing order of their (payment/CPUDemand ratio + their payment/MemoryDemand ratio),** and then selects the sorted jobs one by one until we cannot add more jobs. The stopping criteria is the same as the previous one.

PYTHON

```python
# define the efficiency by adding payment per cpuDemand and payment per memoryDemand
efficiency = [ds.get_jobs().iloc[i]['payment'] / ds.get_jobs().iloc[i]['CPUDemand']
            + ds.get_jobs().iloc[i]['payment'] / ds.get_jobs().iloc[i]['MemoryDemand']
             for i in range(len(ds.get_jobs()))]
```

```
 # add the efficiency to the df
ds.get_jobs()['sorting criterion'] = efficiency
```

We can see that the difference is the second heuristic add an additional payment/MemoryDemand ratio.

2. clearly describe the solutions obtained by the two heuristics. Specifically, for each heuristic, you need to list the "[selected order, job ID, CPU demand, memory demand, payment, sorting criterion]" of each selected job, in the order that they are selected by the heuristic.

The (payment/CPUDemand ratio) heuristic for **small dataset**:

```
smallDS2 = CloudResourceAllocation.constructFromFile(smallFilePath)
get_final_payment_per_cpu_criteria_result(smallDS2,"small cloud resource dataset")
```

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
(payment/cpu ratio) heurstic for small cloud resource dataset :
 Objective value(Total payment): 5647.0
        Total CPU used: 886.0 out of CPU capacity:1000
        Total memory used: 1989.0 out of Memory capacity:2000.
9 selected jobs:
   selected order  Job ID  CPUDemand  MemoryDemand  payment  sorting criterion
4             1.0     5.0        8.0          11.0    381.0          47.625000
7             2.0     8.0       41.0         391.0    907.0          22.121951
9             3.0    10.0       36.0         297.0    531.0          14.750000
6             4.0     7.0       66.0          66.0    675.0          10.227273
8             5.0     9.0      144.0         377.0    962.0           6.680556
2             6.0     3.0      114.0          61.0    724.0           6.350877
0             7.0     1.0      158.0         317.0    836.0           5.291139
1             8.0     2.0      136.0         280.0    607.0           4.463235
5            10.0     6.0      183.0         189.0     24.0           0.131148
```

The (payment/CPUDemand ratio + payment/MemoryDemand ratio) heuristic for **small dataset**:

```
smallDS1 = CloudResourceAllocation.constructFromFile(smallFilePath)
get_final_efficiency_criteria_result(smallDS1,"small cloud resource dataset")
```

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
(payment/cpu ratio + payment/memory) heurstic for small cloud resource dataset :
 Objective value(Total payment): 5647.0
        Total CPU used: 886.0 out of CPU capacity:1000
        Total memory used: 1989.0 out of Memory capacity:2000.
9 selected jobs:
   selected order  Job ID  CPUDemand  MemoryDemand  payment  sorting criterion
4             1.0     5.0        8.0          11.0    381.0          82.261364
7             2.0     8.0       41.0         391.0    907.0          24.441644
6             3.0     7.0       66.0          66.0    675.0          20.454545
2             4.0     3.0      114.0          61.0    724.0          18.219730
9             5.0    10.0       36.0         297.0    531.0          16.537879
8             6.0     9.0      144.0         377.0    962.0           9.232280
0             7.0     1.0      158.0         317.0    836.0           7.928363
1             8.0     2.0      136.0         280.0    607.0           6.631092
5            10.0     6.0      183.0         189.0     24.0           0.258132
```

The (payment/CPUDemand ratio) heuristic for **large dataset**:

```
largeDS2 = CloudResourceAllocation.constructFromFile(largeFilePath)
get_final_payment_per_cpu_criteria_result(largeDS2,"large cloud resource dataset")
```

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
(payment/cpu ratio) heurstic for large cloud resource dataset :
 Objective value(Total payment): 8006.0
        Total CPU used: 1574.0 out of CPU capacity:10000
        Total memory used: 9997.0 out of Memory capacity:10000.
12 selected jobs:
     selected order  Job ID  CPUDemand  MemoryDemand  payment  \
687             1.0   688.0        2.0        1810.0    417.0
682             2.0   683.0        3.0        1031.0    490.0
249             3.0   250.0        2.0        2023.0    282.0
898             4.0   899.0        5.0        1475.0    612.0
297             5.0   298.0        8.0        1859.0    921.0
69              6.0    70.0        7.0        1033.0    755.0
34              7.0    35.0       10.0          60.0    724.0
982             9.0   983.0       21.0         391.0    956.0
410            12.0   411.0       25.0          86.0    907.0
919            16.0   920.0       55.0         220.0    943.0
284           140.0   285.0      509.0           8.0    821.0
528           798.0   529.0      927.0           1.0    178.0

     sorting criterion
687         208.500000
682         163.333333
249         141.000000
898         122.400000
297         115.125000
69          107.857143
34           72.400000
982          45.523810
410          36.280000
919          17.145455
284           1.612967
528           0.192017
```

The (payment/CPUDemand ratio + payment/MemoryDemand ratio) heuristic for **large dataset**:

```
largeDS1 = CloudResourceAllocation.constructFromFile(largeFilePath)
get_final_efficiency_criteria_result(largeDS1,"large cloud resource dataset")
```

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
(payment/cpu ratio + payment/memory) heurstic for large cloud resource dataset :
 Objective value(Total payment): 13057.0
        Total CPU used: 9588.0 out of CPU capacity:10000
        Total memory used: 9976.0 out of Memory capacity:10000.
19 selected jobs:
     selected order  Job ID  CPUDemand  MemoryDemand  payment  \
687             1.0   688.0        2.0        1810.0    417.0
528             2.0   529.0      927.0           1.0    178.0
682             3.0   683.0        3.0        1031.0    490.0
249             4.0   250.0        2.0        2023.0    282.0
898             5.0   899.0        5.0        1475.0    612.0
297             6.0   298.0        8.0        1859.0    921.0
69              7.0    70.0        7.0        1033.0    755.0
284             8.0   285.0      509.0           8.0    821.0
966             9.0   967.0     2184.0           5.0    422.0
34             10.0    35.0       10.0          60.0    724.0
790            11.0   791.0     1332.0          16.0    895.0
982            13.0   983.0       21.0         391.0    956.0
410            14.0   411.0       25.0          86.0    907.0
862            17.0   863.0      119.0          35.0    988.0
151            18.0   152.0     1792.0          20.0    716.0
63             19.0    64.0     1252.0          26.0    884.0
915            20.0   916.0      366.0          38.0    855.0
157            21.0   158.0      366.0          13.0    282.0
712            23.0   713.0      658.0          46.0    952.0

     sorting criterion
687         208.730387
528         178.192017
682         163.808600
249         141.139397
898         122.814915
297         115.620428
69          108.588024
284         104.237967
966          84.593223
34           84.466667
790          56.609422
982          47.968822
410          46.826512
862          36.531092
151          36.199554
63           34.706070
915          24.836066
157          22.462799
712          22.142461
```

3. For each of the two instances, compare the solutions obtained by your two greedy heuristics, and make deep and comprehensive discussions based on your observations.

Foremost, for the cloud resource allocation problem, it is like the knapsack problem that it is based on the **job sorting heuristic.** There is no single greedy heuristic can guarantee to be better then others.

For the small dataset, 2 solutions obtained by my 2 greedy heuristics are exactly the same, which means they have the same total payments , total cpu used, total memory used and the same 9 selected jobs. The only difference is that some Job ID has different selected order, such as the job with the id of 10 has the selected order of 3 and 5 in the (payment/CPUDemand ratio) **heuristic 1**, and the (payment/CPUDemand ratio + payment/MemoryDemand ratio) **heuristic 2** respectively. Due to the difference is too minor, the deep and comprehensive discussion will be made on the large dataset since there are major differences in there.

For the large dataset, the result indicates that the heuristic 2 is better then heuristic 1 since its objective value(total payment) is 13057 which is greater than 8006. For more details, we can see that the total CPU and memory used for heuristic 1 is 9588 and 9976, and for heuristic 2 is 1574 and 9997 out of the total 10000 CPU and memory capacity. These 2 results are expected since the heuristic 1 selects jobs by considering only one dimension, which is the payment/CPUDemand ratio only, whereas the heuristic 2 take care of both dimensions by adding 2 ratios together. Hence, we can see the total CPU and memory used are more balanced for heuristic 2, and the total CPU used is too bias that is pretty low for heuristic 1. It can also be observed through tables of the selected jobs. For the heuristic 1, overall, the CPU Demand is in the ascending order and MemoryDemand is in the descending order, meanwhile the heuristic 2 does not have the obvious order for CPU and memory demand.

Although the heuristic 2 seems better in this cloud resource allocation problem, there is no guarantee that any single greedy algorithm is always better than others, and find the optimal solution. For instance , it is obvious that the solution from the part 1: branch and brand is better than the solutions obtained from greedy heuristic algorithms.

Part 3: Genetic Algorithms (TSP) (30 Marks)

In the report, clearly describe the genetic algorithm framework, chromosome representation, fitness evaluation, initialisation, crossover and mutation operators, and selection schemes.

Genetic Algorithm framework

GA framework can be described as the following pseudocode. It is abstracted from my real code, so hopefully it make sense to you.

PYTHON

```python
def genetic_algorithm_framework(pop_size, stop_criteria) :
    # initialise a  population
    individuals = Initialize_population(pop_size)
    # get the best individual (i.e. fitness evaluation ) from it
    best_individual = get_fittness_from_population(individuals)
    # loop until the max_generations is met (certain number of iterations is reached)
    # and the number of generation_not_improved exceed the threshold
    while not stop_criteria :
        new_generation = [] # construct a new empty population
        # apply elitism by copying the top 5% best individuals to new_generation
        new_generation  = elitism(individuals)
        # repeat until the new generation is full
        while len(new_generation) != len(individuals):
            # tournament select 2 parents from the population
            parent1,parent2 = selection_schemes(individuals)
            # apply the crossover on 2 parents to generate a child
            child1 = apply_crossOver(parent1,parent2)
            # apply the mutation to child, each child has a prob to mutate
            child1.apply_mutation(mutation_rate = 0.1)
            # combined with the local search, 2 opt with probability, locally optimize
            if prob<0.3:
                child1  = local_search_tsp(child1,two_opt_best_improvement)
            elif prob<0.8:
                child1 = local_search_tsp(child1,two_opt_best_improvement)
            # finally , append mutated child into the new generation
            new_generation.append(child1)
        # update the population to the next new_generation
        individuals = new_generation

        # update the best individual from the new generation, evaluate offsprings
        best_individual = get_fittness_from_population(individuals)

    return best_individual
```

For the parameters settings, the **pop_size** indicates that the number of individuals in each generation, I set it to 100. Also, for the stop_criteria parameter setting, there are 2 different ones. One is the loop will be stopped if the certain number of generations(iterations) is met, the other one is if no improvement is found for a number of generations, then stop the iteration.

Chromosome representation

According to [0,1], chromosome is a collection of linked genes that carries genetic information. For the TSP problem, it can also be understand as one potential TSP solution(i.e. a tour), and it can be represented as a list of node IDs(i.e. a permutation of node IDs) for each TSP dataset where each ID is a gene. Also, due to the requirement of the Hamiltonian cycle, each node will be visited exactly once, and the start node id is the same as the end node id.

For example, if a TSP dataset have 4 different locations, then [1,2,3,4,1] is one of the valid chromosome representation that do not have duplicates as well as the head is the same as the tail.

In addition , for my code implementation, Individual class is used to represent a Chromosome, except the tour, it also hold the travel_cost and the fitness of this tour, as we can see from the below screenshot.

```python
class Individual: # for Individual solution
    """
    route of the TSP, which is the chromosome representation, a possible solution to TSP.
    Part of the code is inspired by : https://github.dev/lccasagrande/TSP-GA
    """

    def __init__(self, tour:list, ds:TSP):
        """tour is the set of gene, which is the list of the id of the coord in the TSP.
        For example, [1,2,3,1] is a list of id, do not have the duplicates, the start and end is the same , which is the
        Hamiltonian cycle.

        Args:
            tour(list): a set of ids of the coord in the TSP.
            ds (TSP): the belonged TSP dataset, used to calculate the travel cost and fitness.
        """
        self.tour = tour
        self.ds = ds
        self.__reset_params()

    # def calc_tour_cost(tour, cost_mtx):
    #     return sum([cost_mtx[tour[i], tour[i+1]] for i in range(len(tour)-1)])

    def get_travel_tour_cost(self):
        """
        Calculate the travel cost of the individual.
        """
        self.__travel_cost = sum([self.ds.cost_mtx[self.tour[i], self.tour[i+1]] for i in range(len(self.tour)-1)])
        # due to the hamiltonian cycle, the last and first element is the same
        self.__travel_cost += self.ds.cost_mtx[self.tour[-1], self.tour[0]]
        return self.__travel_cost

    def get_fitness(self):
        """
        Calculate the fitness of the individual.
        It is inspired by : https://github.dev/lccasagrande/TSP-GA
        """
        self.__fitness = 1 / self.get_travel_tour_cost() # normalize the fitness
        return self.__fitness

    def __reset_params(self):
        # assert len(self.tour) > len(self.ds.id_list)
        assert len(set(self.tour)) == len(self.ds.id_list)
        self.__travel_cost = 0
        self.__fitness = 0

        self.get_travel_tour_cost()
        self.get_fitness()
```

fitness evaluation

PYTHON

```python
def get_fitness(self):
        """Calculate the fitness of the individual."""
        self.__fitness = 1 / self.get_travel_tour_cost() # normalize the fitness
        return self.__fitness
```

Fitness evaluation is a gauge of a person's adaptability, according to [0, 1]. This could be the effectiveness of an optimization problem's solution. The aforementioned code sample illustrates how I created my fitness evaluation function. As you can see, it involves normalizing the tour's travel expenses in order to determine which tour is more fit. The lower the cost, the better the fitness.

initialisation

It refers to how to generate the first generation, which is a population of individuals. For my design, I just simply use the random initialisation to generate a population of different tours. In addition, it is worth to mention that my implementation force the permutation for each individual to start from the id 1, which address the symmetry of solution space problem. The corresponding code can be observed below.

PYTHON

```python
@classmethod
def get_a_random_individual(cls,ds:TSP,start_end_node_id:int = 1):
    """
    Create a random individual instance for the TSP.
    The start_end_node is the id of the start and end node, it is used to address the symmetry problem.
    """
    # tempTour =  np.random.permutation(ds.id_list)
    tempTour= np.array(ds.id_list)
    np.random.shuffle(tempTour)
    # remove start_end_node from the tour
    tempTour = np.delete(tempTour, np.where(tempTour == start_end_node_id))
    # add start_end_node to the tour as the head
    tempTour = np.insert(tempTour, 0, start_end_node_id)
    tempTour =  np.append(tempTour, start_end_node_id)
    assert len(tempTour) == len(ds.id_list)+1
    return cls(tempTour.tolist(), ds)
```

```python
def initialize_population(ds:TSP, size) -> list:
    """
    Initialize the population of the TSP.
    Args:
        ds (TSP): the belonged TSP dataset, it is used to calculate the travel cost and fitness.
        size (int): the size of the population.
    """
    population = []
    # repeat until the population is full
    # this initialisation address the issue of symmetry, so use while loop
    while len(population) < size:
        population.append(Individual.get_a_random_individual(ds))
    return population
```

crossover and mutation operators

crossover

It is to produce a offspring by combining parents. The **partially mapped crossover** from the slide[0] is used as my implementation. The idea is that copy a segment from a parent and scan the other parent to fill in the blank and skip the duplicated nodes. Also, to ensure the child is valid (no duplicates, Hamiltonian cycle), several **assert** statements are included within my implementation as the pre- and post-condition check. For conciseness, the code snippet is omitted, but you can check it by yourself if you want.

Mutation

It is a change in a gene sequence, and applied after the crossover. It is designed by me to swap 2 ids from the route with the mutation rate of 10%. In addition, I have also applied the Local Search after the mutation with some probability. They are also called the memetic algorithms that are used to enhance the power of the GA. The deep and comprehensive discussion for LS is discussed later.

Selection schemes

It refers to how to select individuals from the old population as parents to generate the new individual for the next new generation.

The roulette wheel selection is developed and implemented in my code. The main idea is the probability p (a) that each individual a is selected is proportional to his evaluation function f (a). In other words, the parents are selected according to their fitness, the fitter the chromosome is, the more chances it will be selected as the parent to produce the child. The corresponding code is shown below.

PYTHON

```python
@classmethod
def Roulette_wheel_selection_one_from_population(cls,population:list):
    population_copy = copy.deepcopy(population)
    # sort it based on the fitness, high to low
    population_copy.sort(key=lambda x: x.get_fitness(), reverse=True)
    # assign the probablity to each individual according to the fitness
    fitness_sum = sum([x.get_fitness() for x in population_copy])
    fitness_prob = [x.get_fitness() / fitness_sum for x in population_copy]
    # get one individual from the population, based on the probablity
    return np.random.choice(population_copy, p=fitness_prob)
```

There are lots of disadvantages of roulette wheel selection, such as if the fittest individual is much more better than others, then the whole population will be quickly dominated by this individual that other potential better individuals will not be explored. Hence, it causes the high loss of the genetic diversity, and easily be trapped in the local optima.

The tournament selection and the Rank-based selection can be used to address this issue [0]. As you can see from my implementation, I have also tried to sort the population based on fitness first, which is the idea of rank-based selection. However, I cannot really understand how to calculate the cumulative fitness, so it is still the roulette wheel selection, which is a pity.

Others

In addition, I have also implemented the local search(from the given tutorial) and the elitism. The detailed description for each of them is shown below.

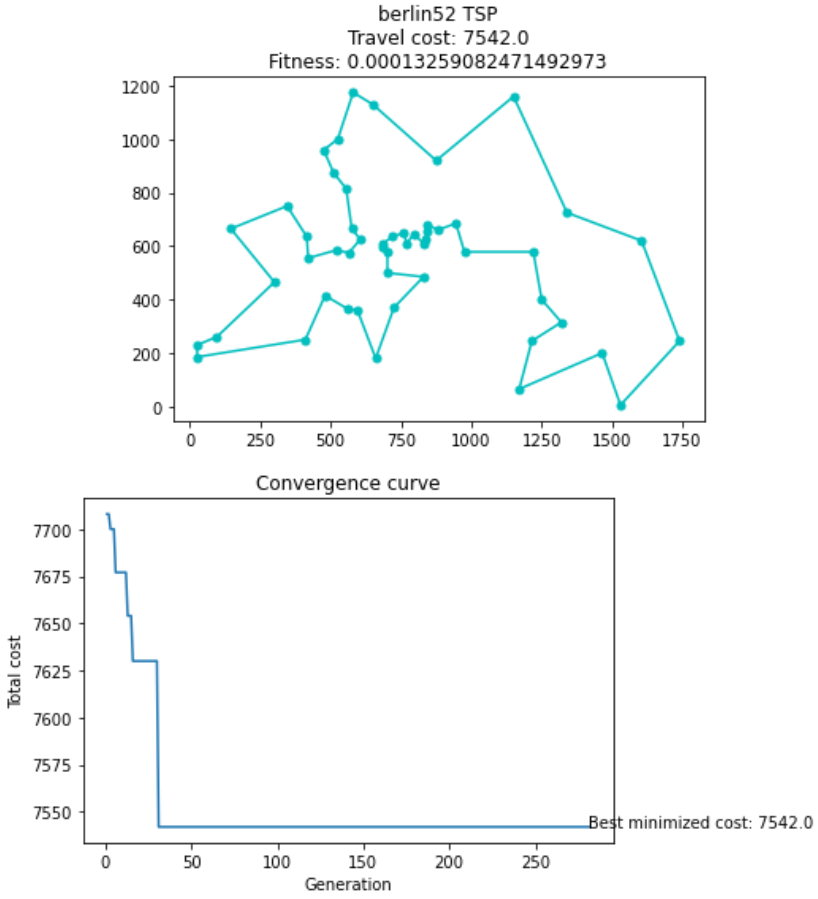
Elitism

It refers to the most fit handful of individuals are guaranteed a place in the next generation, so that the later generation is guaranteed to be equal or with a greater optimum than previous generations. For my design, the top 5% fittest individuals are guaranteed a place in the next generation. The corresponding code snippet is shown below.

```python
new_generation = []
population_size = len(population)
elitism_size = int(0.05 * population_size) # 5% of the population size
# apply the Elitism
for _ in range(elitism_size):
    fittest_ind = Individual.get_fittest_from_population(population)
    new_generation.append(fittest_ind)
```

In the report, draw the convergence curve of the performance, where the x-axis is the generation, and y-axis is the total cost of the best solution in the population in each generation.

The tour route plot and the convergence curve of the performance are displayed below. The minimum travel cost found by my GA is **7542**, which is the same as the most optimal solution for the berlin52 dataset mentioned in [4]. Basically, we can see it converge to the optimal solution within only about 45 generations out of total nearly 300 generations, which is pretty fast.



In the report, compare the results with that obtained by the local search (by running the tutorial code), make deep and comprehensive discussions based on your observations, and draw conclusions.

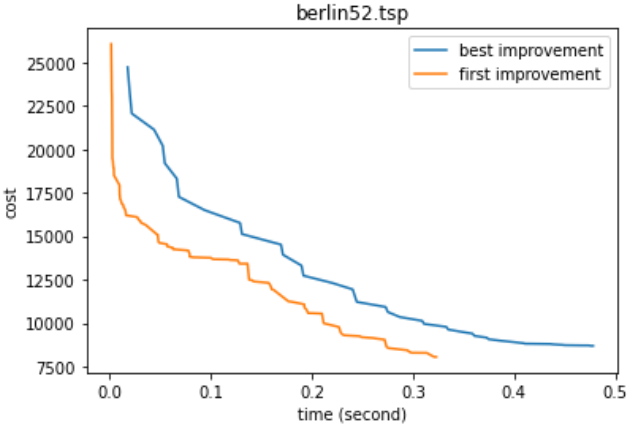
By running the tutorial code on the **berlin52.tsp**, the obtained results are displayed below. Basically, we can observe that the tour costs are **8055** and **8692** found by the two_opt_first_improvement and two_opt_best_improvement, respectively. By comparing it with the GA result (7542), we can observe clearly that the result obtained by the GA is better. **Also, it is worth to mentioned that my GA implementation combine with the Local Search, so it can be called Hybrid Genetic Algorithm(HGA) [3].** The discussion on why GA is better is described below.

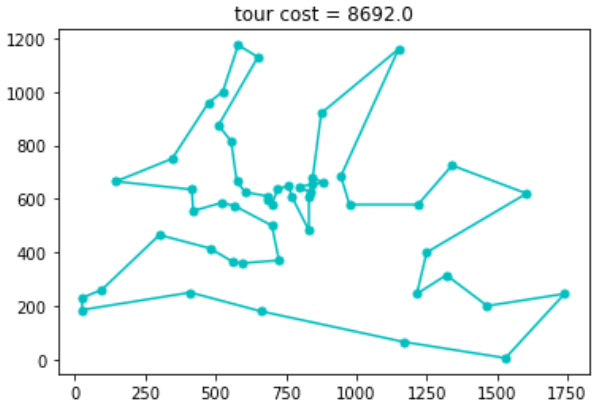
According to the paper[2, 3], GA is a global search algorithm with the ability to find the optimized solution while utilizing minimal resources. It has the stochastic nature that can explore the search space more thoroughly than the local search. The stochastic nature is provided by all the regular GA steps, for example, selected parents are crossed to generate new offspring, and the mutation is included to random swap different gene, which largely increase the exploration capabilities. However, the regular GA also have a major drawback that it is hard to find the better solutions caused by the flat fitness landscape. This conclusion is drawn according to [0, 4] and my personal experiment that the GA is executing without using the Local Search. I have faced the situation that over 80 generations are not improved. And the tour cost of the fittest individual so far is over 10000, which is too far away from the global optimal solution and the local optimal solution obtained from the local search.

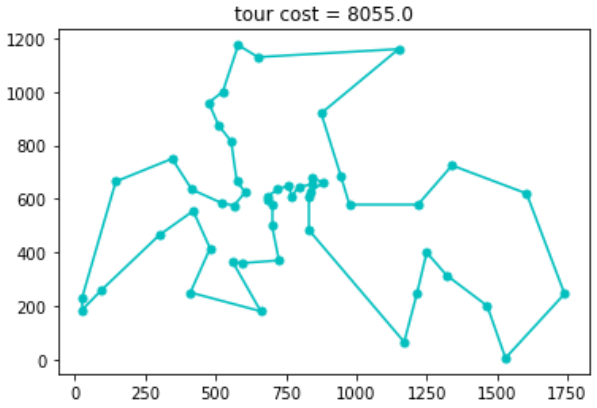
Meanwhile, the local search [1] is a commonly used technique with the guarantee to find the exact local optima. However, it has the major drawback that it is pretty easy to be trapped into the local optima, especially the TSP is a very complex problem with many local optima. It will result in the algorithm to be converged locally bur not globally. For example, the 2-opt move operator is a very basic

operator that only 2 new edges are changed, which cause only the small size neighbourhood around the current solution is explored. Hence, it is easy to be trapped into the local optima.

In conclusion, it is a good complement to combine the GA with the Local Search since both the search capability and the search space exploration thoroughly are ensured. By applying the Local search on the regular GA, each mutated child candidate is optimized locally, which ensures the exact local optima is converged, and it will be used in the next generation. Meanwhile, only using the regular GA will result in a situation that it is hard to converge, and Local Search will result in a situation that it is easy to be trapped in the local optima.







Part 4: Genetic Programming Hyper-heuristics (40 marks)

Run the GP algorithm on the berlin52.tsp instance with 3 different random seeds. For each run, report the following in the report:

The best GP program of the run (the tree);

Draw the TSP solution generated by the best GP program and its total cost;

For this part 4, only the code for the nearest neighbour heuristic part has been successfully implemented. For the GA hyper-heuristic part, the DEAP framework is used, but I do not know where the bug is. The generated heuristic cannot really be used as the scoring function, which cause it cannot decide which node to visit next.

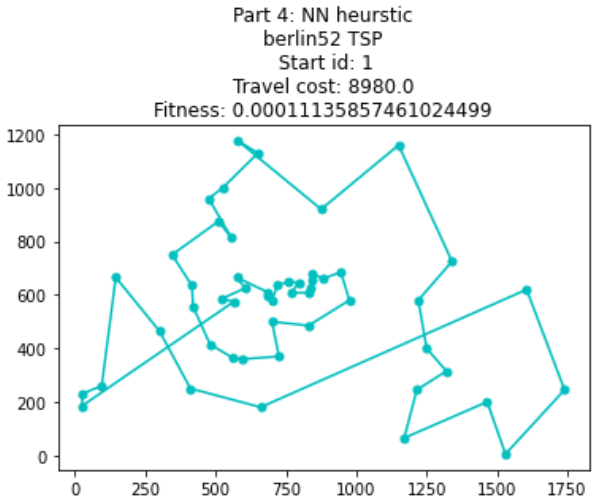
Hence, for this part, I have written some discussions based on materials [0,6,7]. For the design of my GP algorithm, such as function and terminal sets, you can look at my source code. Hopefully it can give me some marks.

The following screenshot displays the nearest neighbour heuristic implementation. We can see that the run_tsp_for_given_heurstic_0 function accept an **heuristicFunction argument**, which is the part that we want to evolve since it has a direct impact on the TSP.

```python
def get_nearest_negibour(ds:TSP, current_node_id, visited_nodes) -> int:
    """
    Get the nearest negibour of the city_index.
    Args:
        ds (TSP): the belonged TSP dataset, used to calculate the travel cost
    """
    nearest_negibour = None
    min_cost = float('inf')
    for id in ds.id_list:
        if id in visited_nodes: continue
        # calculate the travel cost between the current node and the other node
        cost = ds.cost_mtx[current_node_id, id]
        if cost < min_cost:
            min_cost = cost
            nearest_negibour = id
    return nearest_negibour # the id

def run_tsp_for_given_heurstic_0(heuristicFunction,ds:TSP,startNodeID:int=1):
                                 #heuristicFunction:Callable[[TSP,int,list],int]=get_nearest_negibour):
    """For solving the TSP problem. Enforce to start from node 1, always expand by choosing the nearest
    negibhour to go.
    `It is the fitness evaluation function that following 5 steps from the handout`
    Args:
        ds (TSP): the TSP datase instance, such as a280.tsp, berlin52.tsp.
    """
    # start from node 1
    current_node = startNodeID
    # initialize the tour
    tour = [current_node]
    # loop until the tour is complete
    while len(tour) < len(ds.id_list):
        # get the expanded node from the current node
        node_to_expand_next = heuristicFunction(ds, current_node, visited_nodes=tour)
        # add it to the tour
        tour.append(node_to_expand_next)
        # update the current node to the nearest negibour
        current_node = node_to_expand_next
    # add the first node to the tour, to from the hamiltoniam cycle
    tour.append(startNodeID)
    ind = Individual_part3(tour, ds)
    cost = ind.get_travel_tour_cost()
    return  cost, ind, tour,
```

Compare with the solution generate by the nearest neighbour heuristic and make discussions.



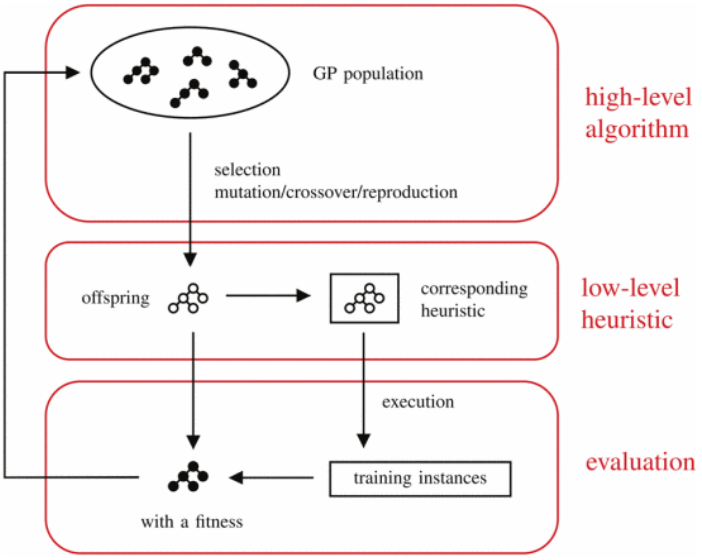
The above figure display the solution generated by the nearest neighbour heuristic. Although I fail to perform the GP experiment, the paper[6] mention some experiments that demonstrate the generated heuristics are outperforming existing ones. To explain why, some discussions are made below, and they are inspired from the paper [0,6,7].

First of all, the heuristics are required for solving CO problems, hence, a wide range of heuristics has been developed in order to obtain the good quality solutions within the reasonable time on large scale CO problems [0,6]. However, in order to design the proper heuristic for the given CO problem, the domain human expert needs to be involved to spend lots of time to perform the refinement iteratively. In other words, it has the limitation that heuristics are problem specified.

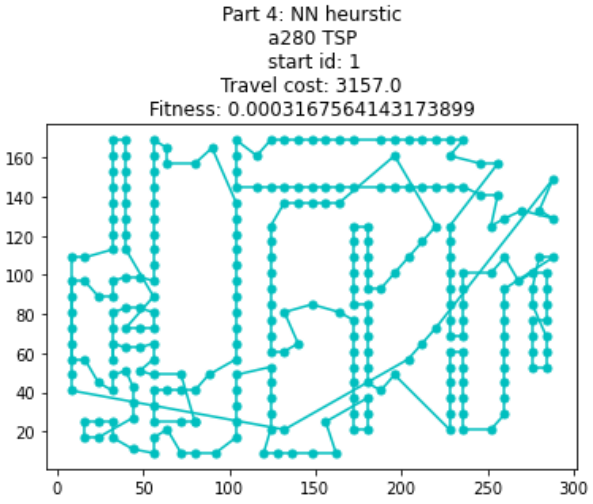
A good example is the given nearest neighbour greedy heuristic for TSP problem. As we can see from the the solution generate by the nearest neighbour heuristic displayed above, it is clearly that it has the potential issue that the node might be left isolated and will be linked through a very long path, so it definitely needs the human experts to refine.

Meanwhile, the presence of the hyper-heuristic address this limitation by automatically selecting / generating the heuristic. That means, the search space for GP Hyper-heuristics is not the solution space that will finally yield the solution any more. Instead, the GP hyper-heuristic is in the higher level that **a population of individuals representing heuristics is evolved**. The below figure [6] indicates this process, and it will not stop until the perfect program, the heuristic, is evolved.

Hence, we can see how GP hyper-heuristic evolve the heuristic, and that is the reason why the generated heuristic yields the better result than the nearest neighbor heuristic.



Apply the best GP program to the `a280.tsp` instance, and draw the generated solution and its total cost. Compare the solution with that generated by the nearest neighbour heuristic and make deep and comprehensive discussions based on your observations.



I think the generated solution will be reasonably good, and better than the above solution generated by the Nearest Neighbor heuristic solution. As mentioned above, heuristics have the limitation that it is problem specified. However, a280 and berlin52 are both the same TSP problem, just different instances. Therefore, I think the generated heuristic might be bias to the berlin52 instance, but it can still be applied on the a280.tsp instance and yield the reasonably good total costs.

Part 4: Question (10 marks)

Discuss about the advantages and disadvantages of the following techniques for scheduling and combinatorial optimization problems, and how to choose a proper technique to solve a combinatorial optimization problem.

Mathematical programming (e.g., Mixed Integer Linear Programming)

Branch and Bounds is a method that we learnt and used for this assignment. The main idea is to build the linear model of the CO problem by handling the non-linear constraints and objective functions, so that it can be transformed to the relaxed linear model, which means it is easier to solve.

Pros:

1. it can speed up the exhaustive search process, and the optimal solution can be obtained much faster than the exhaustive search.
   - In the example of the Branch and Bound,we can see that it is achieved by early pruning sub-branches.
2. The mathematical programming methods can **guarantee optimality**.

Cons:

1. The computational complexity is still exponentially expensive, which means if the problem size is bigger, then it will be very slow. It cannot solve complex large problems.
2. Efficiency depends on heuristics.

Greedy heuristics

Pros:

1. The speed is fast, a solution can be generated quickly .
    - It address the disadvantage of the Mathematical programming that even the problem size is pretty large, we can still obtain the near-optimal solutions in a short time by using the heuristics. That is,
2. the solution quality is reasonably good.
3. It is intuitive, easy to implement.

Cons:

1. Optimal solution is not guaranteed.
2. Performance highly depends on the heuristic.
3. Sometimes the solution quality can be very poor.
    - For example, the solution quality is poor for the 1st heuristic that I design in part 2 .

Genetic algorithms

For this one, the pros and cons only applied on the regular GA, that means it is not applied on the Hybrid Genetic Algorithms that combine with the Local Search.

Pros:

1. A number of optimal solutions is capable to be found rather than a single solution [2]
2. GA is capable to explore the search space more thoroughly.[2]
3. Unlike Local search, GA is less dependent on the starting point which refer to the initialisation[2].
4. GA does not require the neighbourhood definition.
5. If the optimization problem can be described with the chromosome encoding, then GA can be used to solve.
6. GA can be used to solve the problem with multiple solutions.
7. multi-dimensional, non-differential, non-continuous, and even non-parametrical problems can be solved by using GA[2].

Cons: As the heuristic method [2], GA has the following disadvantages :

1. It is hard to reach the global optima.
2. It requires long time to optimize.
3. It cannot be applied to certain problems, such as variant problem.[2]
    - It is because the bad chromosomes will be generated and result in poor fitness.

Genetic programming hyper-heuristics

Pros:

1. The algorithm can be designed or configured automatically from scratch.
2. It has the advantage to handle multiple heuristics to tackle problem instances [6].

Cons: I did not find the disadvantages from online. The Genetic programming hyper-heuristic is much more powerful than the Genetic Algorithm that is used to design programs to write programmes. I think it is the best technique that I have seen so far.

Probably, the potential disadvantages are the same as the Genetic Algorithms since it is just the enhanced version of the Genetic Algorithms.

How to choose a proper technique to solve a combinatorial optimization problem

I think it depends on the different problems and situations.

If the CO problem can be represented as the mathematical linear model, then probably Mathematical programming should be chosen since it can guarantee the optimality, and linear models are easier to solve.

If the time is in urgent, and there is no optimality requirements, then the greedy heuristics should be chosen since it is intuitive and it can generate a reasonably good solution quickly.

If the CO problem can be described with the chromosome encoding, and it is a kind of blackbox that **nobody has explored deeply**, then probably the Genetic Algorithms and Genetic Programming Hyper-heuristics are the most suitable ones. It is because search space is guaranteed to be explored thoroughly, and we do not need the human experts to involve. Especially nobody knows how to address this CO problem.

Bibliography

[0] Slides and Jupyter Notebooks

[1] S. Abenga, “Genetic Algorithms,” *The Andela Way*, Nov. 14, 2017. https://medium.com/the-andela-way/on-genetic-algorithms-and-their-application-in-solving-regression-problems-4e37ac1115d5

[2] J. Tumuluru and R. McCulloch, “Application of Hybrid Genetic Algorithm Routine in Optimizing Food and Bioengineering Processes,” *Foods*, vol. 5, no. 4, p. 76, Nov. 2016, doi: 10.3390/foods5040076.

[3] Y. Deng, Y. Liu, and D. Zhou, “An Improved Genetic Algorithm with Initial Population Strategy for Symmetric TSP,” *Mathematical Problems in Engineering*, vol. 2015, p. e212794, Oct. 2015, doi: 10.1155/2015/212794.

[4] “Symmetric TSPs,” *comopt.ifi.uni-heidelberg.de*. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html

[5] “A Survey of Genetic Programming and Its Applications,” *KSII Transactions on Internet and Information Systems*, vol. 13, no. 4, Apr. 2019, doi: 10.3837/tiis.2019.04.002.

[6] G. Duflo, E. Kieffer, M. R. Brust, G. Danoy, and P. Bouvry, “A GP Hyper-Heuristic Approach for Generating TSP Heuristics,” *IEEE Xplore*, May 01, 2019. https://ieeexplore.ieee.org/document/8778254