TP3-AI

Zhenxi Li(1883294) Jinling Xing(1915481)

December 3, 2017

1 Question-ACO&RL

1.1 Description

How do you associate the ant colony optimization algorithm with reinforcement learning?

1.2 Explanation

It is well known that the ant by means of pheromone to construct a path to finding food. Ants deposit some amount of pheromone when walking, and each follows its direction according the total amount of pheromone deposited.

Since the initial values of tij (the pheromone value between edge[i,j]) are the same for all the edges, the ants are first free to move. The good edges will be used by a lot of ants and, therefore, will receive a greater amount of pheromone.

This is a reinforcement learning system where reinforcement is applied by modifying the force (pheromone) connections between the vertices.

Two important principles in the process of ant colony optimization:

1. local pheromone update.
2. global pheromone update.

The pheromone updating is that ant simulates the change of pheromone quantity according to the pheromones and evaporated pheromones left by the other ant on the visited vertices (cities), so that the better solution gets more enhancement. This will form a positive feedback mechanism for Reinforcement Learning.

2 getNextCity

2.1 Description

This function returns the next city to be added into solution using the expressions (1) and (2) in the tp3-fall2017.

$$j = \begin{cases} \arg\max_{j \in \overline{V}(S_k)} \left\{ \dfrac{\tau(e_{ij})}{c(e_{ij})^{\beta}} \right\}, \text{if } q \leq q_0 \\ \text{randomly selected in the probability distribution } D \end{cases} \tag{1}$$

$$p_k(e_{ij}) = \begin{cases} \dfrac{\tau(e_{ij})/c(e_{ij})^{\beta}}{\sum\limits_{l \in \overline{V}(S_k)} \tau(e_{il})/c(e_{il})^{\beta}}, \text{if } j \in \overline{V}(S_k) \\ 0, \text{otherwise} \end{cases} \tag{2}$$

2.2 Code

```
def get_next_city(self, sol):
    q = ((float)(random.randint(0,100)))/100
    tmpI = SOURCE
    if (len(sol.not_visited) == 1):
        return SOURCE
    if (len(sol.visited) > 0) :
        tmpI = sol.visited[len(sol.visited)-1]
    ratioDic = {}
    for j in sol.not_visited:
        if (j == SOURCE):
            continue
        tmpRatio = 0
        tmpRatio = ((float(self.pheromone[tmpI,j]))/
         (self.graph.costs[tmpI,j]**self.parameter_beta)
        ratioDic[j] = tmpRatio
    if q <= self.parameter_q0:
        return max(ratioDic, key=ratioDic.get)
    else:
        mulRatio = {}
        for key, value in ratioDic.items():
            mulRatio[key] = (int)(round(value*1000))
        tmpRan = random.randint(0, sum(mulRatio.values()))
        for key, value in mulRatio.items():
            if (tmpRan <= value):
                return key
            tmpRan = tmpRan-value
```

2.3 Result

When we tested code using the test function testNextCity() provided in the *ACO_test.py*, the test result is OK.

2.4 Explanation

First of all, q is a random number uniformly distributed in [0...1] we did it use the $q = ((float)(random.randint(0, 100))$ in the code. The next step we did is that we checked whether the graph is already visited or if the graph not empty.
Second, we used a for loop to calculate the formula (1) and the used an if-else clause to achieve the both formulas (1) and (2). If q is smaller than q_0, we returned the index of the max value, otherwise we calculated the probability according to the formula (2).

3 heuristic2opt

3.1 Description

The heuristic2opt required us to apply the local search 2-opt in the solution sol.

3.2 Code

```
def heuristic2opt(self, sol):
    for i in range(0, len(sol.visited)):
        for j in range(0, len(sol.visited)):
            if i == j or (i-1+len(sol.visited))%len(sol.visited) == j
             or (i+1)%len(sol.visited) == j:
                continue
            oldCost = self.graph.costs[sol.visited[i], sol.visited[(i+1)%len(sol.visited)]] +
            self.graph.costs[sol.visited[j], sol.visited[(j+1)%len(sol.visited)]]
```

```
newCost = self.graph.costs[sol.visited[i],
sol.visited[j]] +  self.graph.costs[sol.visited[(i+1)%len(sol.visited)],
 sol.visited[(j+1)%len(sol.visited)]]
if (newCost < oldCost):
    sol.inverser_ville(i, j)
    sol.cost = sol.cost + newCost - oldCost
```

3.3 Result

When we tested code using the test function testHeuristic2opt() provided in the $ACO_test.py$, the test result is OK.

3.4 Explanation

We used a while clause to check the improvement and we used double-for loop to calculate the newCost and oldCost. If the newCost is smaller than the oldCost to swap the cities and re-calculated the cost until the improvement is zero.

4 globalUpdate

4.1 Description

This question required us to perform the global updating step using the best solution found, as showed in (3) and (4).

$$\tau(e_{ij}) = (1-\rho)\tau(e_{ij}) + \rho\Delta\tau(e_{ij}) \tag{3}$$

$$\Delta\tau(e_{ij}) = \begin{cases} \frac{1}{L_{gb}}, \text{if } e_{ij} \in \text{global-best tour,} \\ 0, \text{otherwise.} \end{cases} \tag{4}$$

4.2 Code

```
def global_update(self, sol):
    bestLen = sol.cost
    for i in range(0, self.graph.N):
        for j in range(i, self.graph.N):
            indexI = sol.visited.index(i)
            indexJ = sol.visited.index(j)
            if ((i == SOURCE and indexJ == SOURCE) or (abs(indexI-indexJ) == 1)):
                self.pheromone[i,j] =
                 (1-self.parameter_rho)*self.pheromone[i,j]+
                 self.parameter_rho*(float(1)/bestLen)
                self.pheromone[j,i] = self.pheromone[i,j]
            else:
                self.pheromone[i,j] =
                 (1-self.parameter_rho)*self.pheromone[i,j]
                self.pheromone[j,i] = self.pheromone[i,j]
```

4.3 Result

When we tested code using the test function testGlobalUpdate() provided in the $ACO\_test.py$, the test result is OK.

4.4 Explanation

We used double-for loops to update the pheromone, and we used if-else clause to perform the formulas (3) and (4). $if((i == 0 and indexJ == 0) or (abs(indexI - indexJ) == 1))$, when the

condition is satisfied, we increased the pheromone levels associated with the best solution found. Otherwise, we just needed to decrease all the pheromone values through pheromone evaporation.

5 localUpdate

5.1 Description

This function required to perform the local updating step using in the solution sol, as showed in (5).

$$\tau(e_{ij}) = (1 - \varphi)\tau(e_{ij}) + \varphi\tau_0(e_{ij}). \tag{5}$$

5.2 Code

```
def local_update(self, sol):
    start = SOURCE
    end = -1
    for i in range(0, self.graph.N):
        end = sol.visited[i]
        self.pheromone[start,end] =
        (1-self.parameter_phi)*self.pheromone[start,end]+
         self.parameter_phi*self.pheromone_init[start,end]
        self.pheromone[end,start] = self.pheromone[start,end]
        start = end
```

5.3 Result

When we tested code using the test function testLocalUpdate() provided in the *ACO_test.py*, the test result is OK.

5.4 Explanation

We used a for loop to perform the local pheromone update, after one city is visited. First,we did the pheromone according to the formula (5), and we also need to mention that the pheromone update is symmetry, such as $self.pheromone[end, start] = self.pheromone[start, end]$.

6 runACO

6.1 Description

This question required to run the ACO algorithm and return the best solution found. The parameter numberIteration is the maximum number of iterations. Here you must implement the work ow of the ACO algorithm, i.e., creation of K ants; construction of the solution; execution of the local search routine; and updating the pheromone.

6.2 Code

```
def runACO(self, maxiteration):
    startTime = time.time()
    for ite in range(0, maxiteration):
        for k in range(0, self.parameter_K):
            tmpSol = Solution(self.graph)
            start = SOURCE
            while (len(tmpSol.not_visited) != 0):
                nextNode = self.get_next_city(tmpSol)
                tmpSol.add_edge(start, nextNode)
                start = nextNode
            self.local_update(tmpSol)
```

```
            if tmpSol.cost < self.best.cost:
                self.best = tmpSol
        self.heuristic2opt(self.best)
        self.global_update(self.best)
    endTime = time.time()
    return (endTime-startTime)
```

6.3 Result

When we tested code, the code is working fine and you get the optimal solution.

6.4 Explanation

At each iteration, each ant do the while loop to find the next city and do the local update, and do the global update and the 2-opt heuristic optimization.

7 Tuning the parameters

7.1 Description

As required by many artificial intelligence methods, it is first necessary to tune the set of parameters so that the algorithm can achieve its best performance. We need to test five sets of parameters. A simple procedure to tune a parameter ρ belongs to P is fix the others $|P| - 1$ parameters, and run multiple experiments changing the value of p in order to observe the algorithm convergence. The range of parameters are q0[0,1] and iteration is 0.1, beta[0,3] and iteration is 0.5, rho[0,1] and iteration is 0.1, phi[0,1] and iteration is 0.1, and K[5,40] and iteration is 5.

7.2 Code

```
default_q0 = 0.9
default_beta = 2
default_rho = 0.1
default_phi = 0.1
default_K = 10
default_iteration = 1000

tune_default_time = None
tune_default_cost = None

tune_q0_time = []
tune_q0_cost = []
tune_beta_time = []
tune_beta_cost = []
tune_rho_time = []
tune_rho_cost = []
tune_phi_time = []
tune_phi_cost = []
tune_K_time = []
tune_K_cost = []

def thread_tune_default():
    global tune_default_time, tune_default_cost
    global default_q0, default_beta, default_rho
    global default_phi, default_K, default_iteration

    default_time = []
```

```
    default_cost = []
    print("Start default...")
    for i in range(1,6):
        tmpAco = ACO(default_q0, default_beta, default_rho, default_phi, default_K, 'qatar')
        tmpTime = tmpAco.runACO(default_iteration)
        tmpCost = tmpAco.best.cost
        default_time.append(tmpTime)
        default_cost.append(tmpCost)
        print(tmpTime)
        print(tmpCost)
    tune_default_time = np.mean(default_time)
    tune_default_cost = np.mean(default_cost)
    print("tune_default_time: " + str(tune_default_time))
    print("tune_default_cost: " + str(tune_default_cost))


def thread_tune_q0():
    global tune_q0_time, tune_q0_cost
    global default_q0, default_beta, default_rho
    global default_phi, default_K, default_iteration

    ### tuning q0
    q0_start = 0
    q0_step = 0.1
    q0_end = 1

    print("Start q0...")
    while q0_start <= q0_end:
        cur_q0_time = []
        cur_q0_cost = []
        for i in range(1,6):
            tmpAco = ACO(q0_start, default_beta, default_rho, default_phi, default_K, 'qatar')
            tmpTime = tmpAco.runACO(default_iteration)
            tmpCost = tmpAco.best.cost
            cur_q0_time.append(tmpTime)
            cur_q0_cost.append(tmpCost)
            print(tmpTime)
            print(tmpCost)
        tune_q0_time.append(np.mean(cur_q0_time))
        tune_q0_cost.append(np.mean(cur_q0_cost))
        q0_start = q0_start+q0_step
    print("tune_q0_time: " + str(tune_q0_time))
    print("tune_q0_cost: " + str(tune_q0_cost))


def thread_tune_beta():
    global tune_beta_time, tune_beta_cost
    global default_q0, default_beta, default_rho
    global default_phi, default_K, default_iteration

    ### tuning beta
    beta_start = 0
    beta_step = 0.5
    beta_end = 3

    print("Start beta...")
    while beta_start <= beta_end:
        cur_beta_time = []
```

```
        cur_beta_cost = []
        for i in range(1,6):
            tmpAco = ACO(default_q0, beta_start, default_rho, default_phi, default_K, 'qatar')
            tmpTime = tmpAco.runACO(default_iteration)
            tmpCost = tmpAco.best.cost
            cur_beta_time.append(tmpTime)
            cur_beta_cost.append(tmpCost)
            print(tmpTime)
            print(tmpCost)
        tune_beta_time.append(np.mean(cur_beta_time))
        tune_beta_cost.append(np.mean(cur_beta_cost))
        beta_start = beta_start+beta_step
    print("tune_beta_time: " + str(tune_beta_time))
    print("tune_beta_cost: " + str(tune_beta_cost))


def thread_tune_rho():
    global tune_rho_time, tune_rho_cost
    global default_q0, default_beta, default_rho
    global default_phi, default_K, default_iteration

    ### tuning rho
    rho_start = 0
    rho_step = 0.1
    rho_end = 1

    print("Start rho...")
    while rho_start <= rho_end:
        cur_rho_time = []
        cur_rho_cost = []
        for i in range(1,6):
            tmpAco = ACO(default_q0, default_beta, rho_start, default_phi, default_K, 'qatar')
            tmpTime = tmpAco.runACO(default_iteration)
            tmpCost = tmpAco.best.cost
            cur_rho_time.append(tmpTime)
            cur_rho_cost.append(tmpCost)
            print(tmpTime)
            print(tmpCost)
        tune_rho_time.append(np.mean(cur_rho_time))
        tune_rho_cost.append(np.mean(cur_rho_cost))
        rho_start = rho_start+rho_step
    print("tune_rho_time: " + str(tune_rho_time))
    print("tune_rho_cost: " + str(tune_rho_cost))


def thread_tune_phi():
    global tune_phi_time, tune_phi_cost
    global default_q0, default_beta, default_rho
    global default_phi, default_K, default_iteration

    ### tuning phi
    phi_start = 0
    phi_step = 0.1
    phi_end = 1

    print("Start phi...")
    while phi_start <= phi_end:
        cur_phi_time = []
```

```
        cur_phi_cost = []
        for i in range(1,6):
            tmpAco = ACO(default_q0, default_beta, default_rho, phi_start, default_K, 'qatar')
            tmpTime = tmpAco.runACO(default_iteration)
            tmpCost = tmpAco.best.cost
            cur_phi_time.append(tmpTime)
            cur_phi_cost.append(tmpCost)
            print(tmpTime)
            print(tmpCost)
        tune_phi_time.append(np.mean(cur_phi_time))
        tune_phi_cost.append(np.mean(cur_phi_cost))
        phi_start = phi_start+phi_step
    print("tune_phi_time: " + str(tune_phi_time))
    print("tune_phi_cost: " + str(tune_phi_cost))


def thread_tune_K():
    global tune_K_time, tune_K_cost
    global default_q0, default_beta, default_rho
    global default_phi, default_K, default_iteration

    ### tuning K
    K_start = 5
    K_step = 5
    K_end = 40

    print("Start K...")
    while K_start <= K_end:
        cur_K_time = []
        cur_K_cost = []
        for i in range(1,6):
            tmpAco = ACO(default_q0, default_beta, default_rho, phi_start, default_K, 'qatar')
            tmpTime = tmpAco.runACO(default_iteration)
            tmpCost = tmpAco.best.cost
            cur_K_time.append(tmpTime)
            cur_K_cost.append(tmpCost)
            print(tmpTime)
            print(tmpCost)
        tune_K_time.append(np.mean(cur_K_time))
        tune_K_cost.append(np.mean(cur_K_cost))
        K_start = K_start+K_step
    print("tune_K_time: " + str(tune_K_time))
    print("tune_K_cost: " + str(tune_K_cost))
```

7.3 Result

For each parameter p, you must report two plots: p*best solution cost; and p*execution time.There is a set of initial values that you we can use to fix the parameter before optimizing it:

```
default_q0 = 0.9
default_beta = 2
default_rho = 0.1
default_phi = 0.1
default_K = 10
default_iteration = 1000
```

The default time and cost are as follows:

$tune\_default\_time$: 1135.85068808
$tune\_default\_cost$: 10221.20144

The result of each parameter we shown as plots, for each parameter the first plot is average best solution cost and the second plot is the average execution time, we attached these plots as below.

From these ten plots, we can get the optimal parameters are: q0=1, beta=2, rho= 0.2, phi=1.0, K=20, considering their best solution cost.

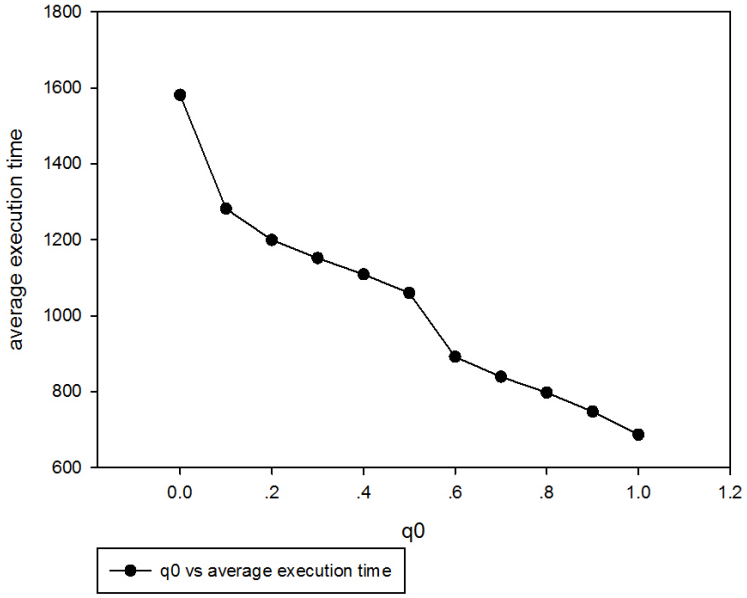

Figure 1: The graph of q0 and average execution time

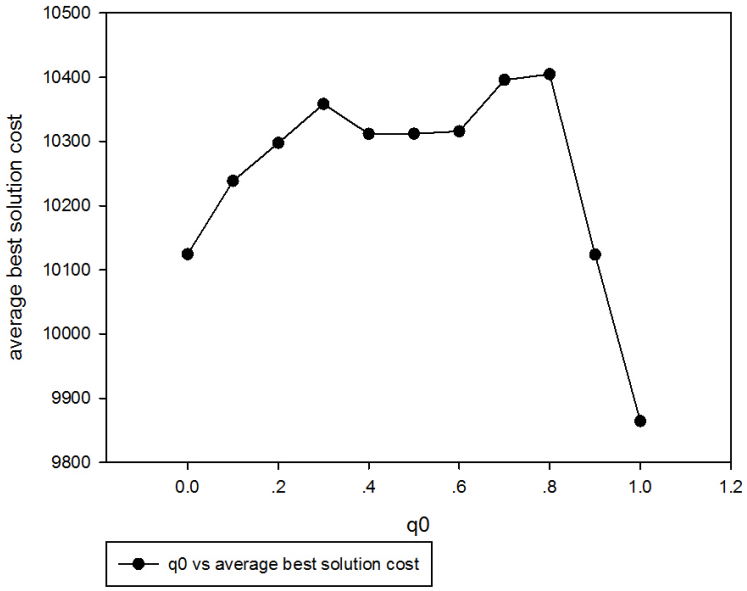

Figure 2: The graph of q0 and average best solution cost

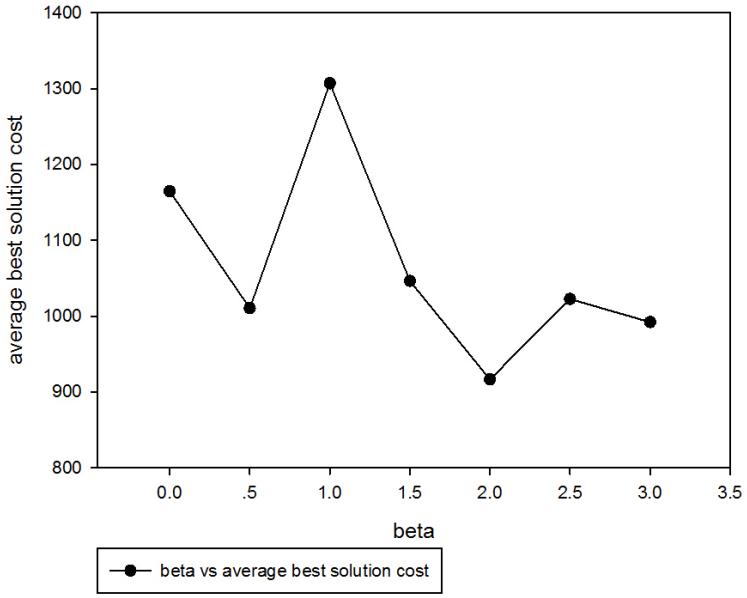

Figure 3: The graph of beta and average execution time

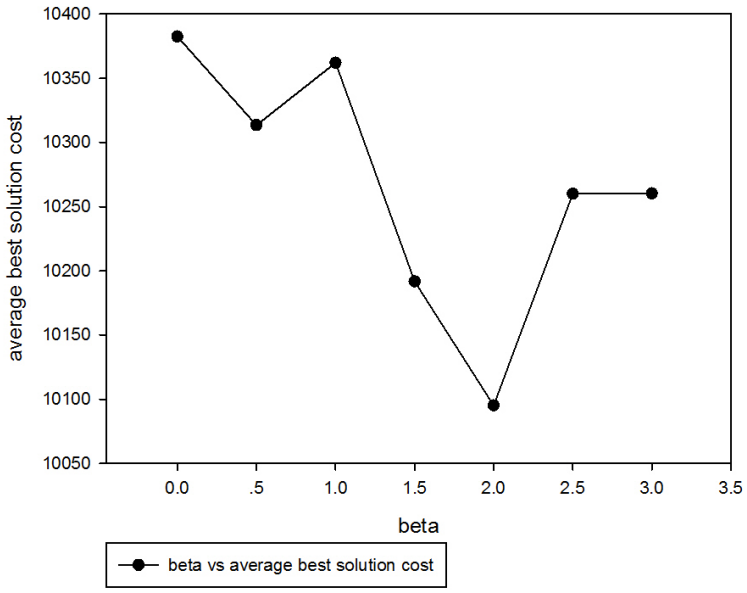

Figure 4: The graph of beta and average best solution cost

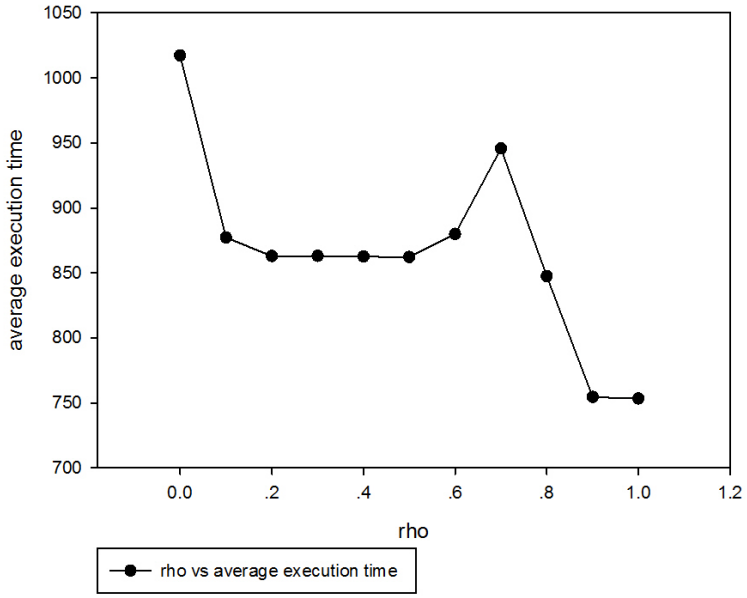

Figure 5: The graph of rho and average execution time

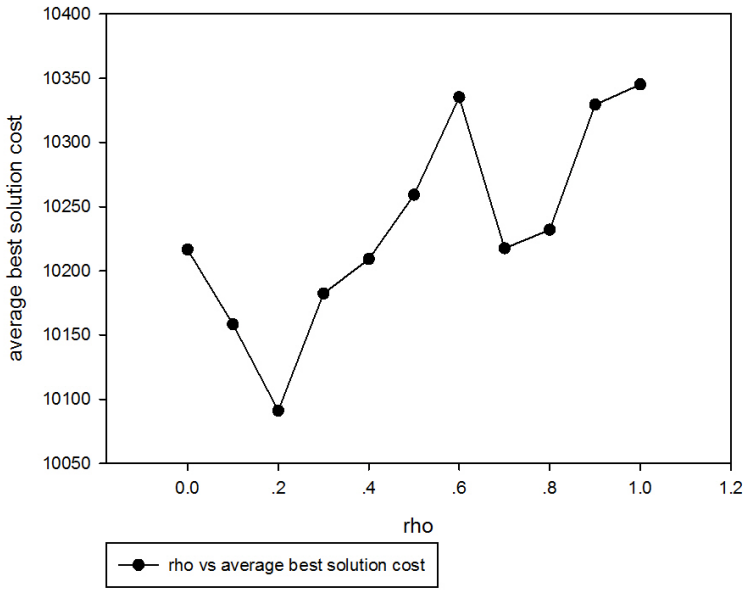

Figure 6: The graph of rho and average best solution cost

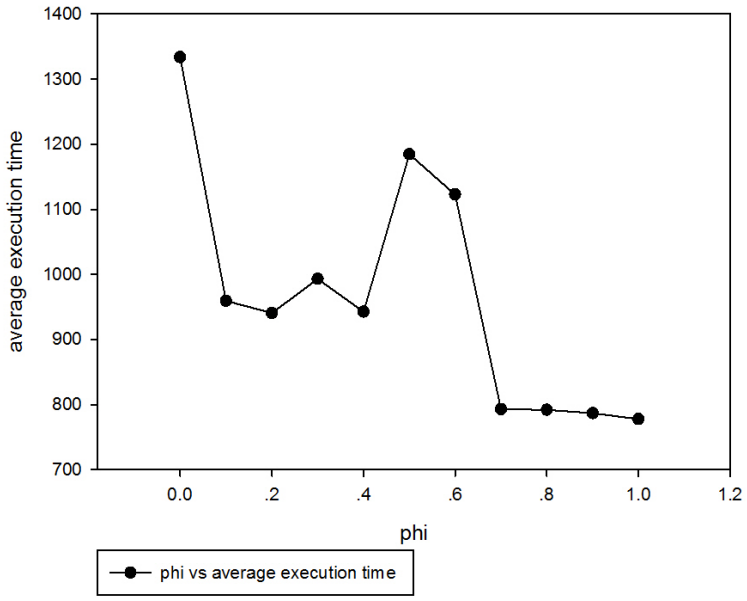

Figure 7: The graph of phi and average execution time

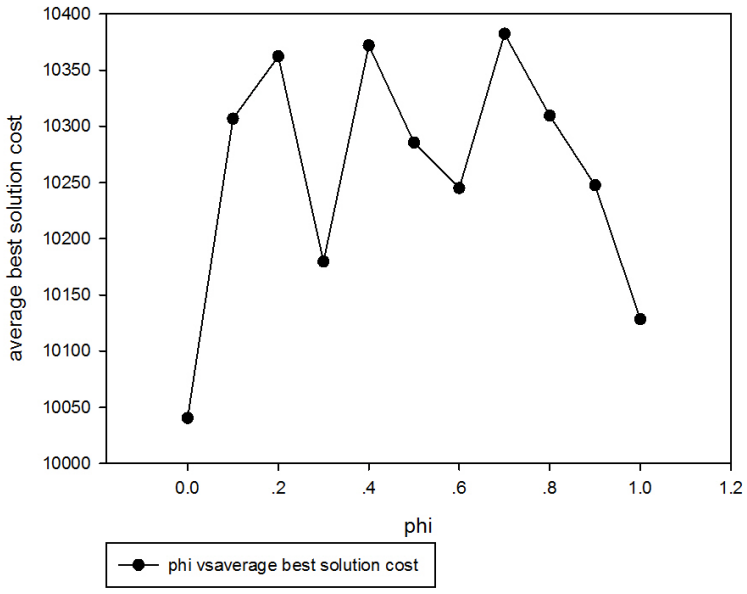

Figure 8: The graph of phi and average best solution cost

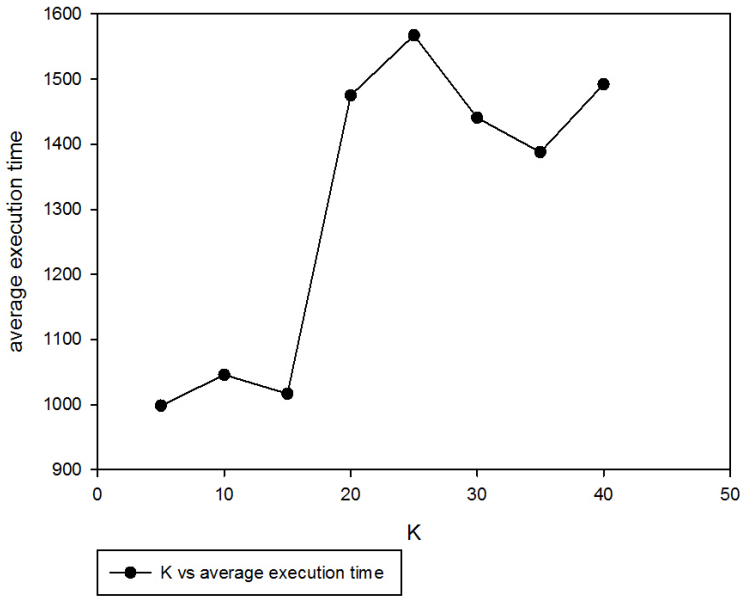

Figure 9: The graph of K and average execution time

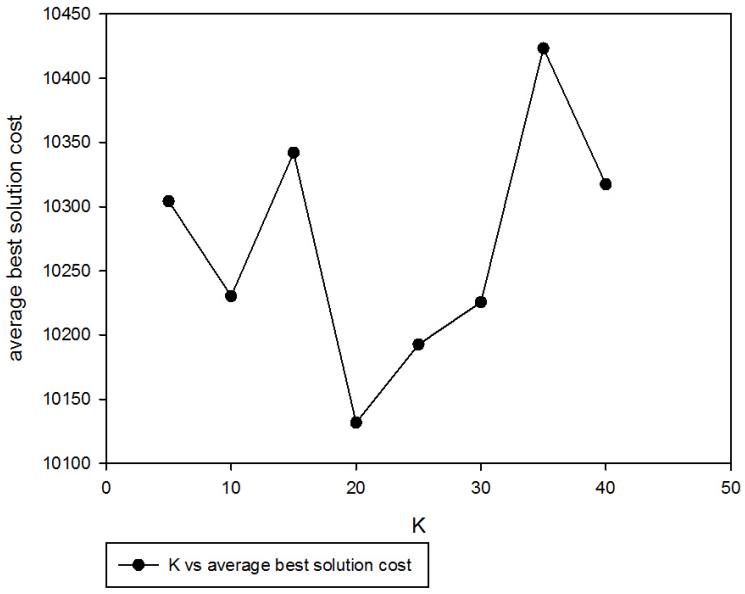

Figure 10: The graph of K and average best solution cost

8 Running your algorithm

8.1 Description

Using our set of optimized parameters, we run our algorithm for the two national instances of TSP showing in the Table 1 and Table 2. We summarized our results in the Table 3 below. To print the result for optimizing Uruguay and Canada dataset, we add a new function to ACO:

8.2 Code

```
def optimizeGap(self, maxiteration, OPT):
    startTime = time.time()
    lastBest = -1
    lastImp = 0
    for ite in range(0, maxiteration):
        for k in range(0, self.parameter_K):
            tmpSol = Solution(self.graph)
            start = SOURCE
            while (len(tmpSol.not_visited) != 0):
                nextNode = self.get_next_city(tmpSol)
                tmpSol.add_edge(start, nextNode)
                start = nextNode
            self.local_update(tmpSol)
            if tmpSol.cost < self.best.cost:
                self.best = tmpSol
        self.heuristic2opt(self.best)
        self.global_update(self.best)

        if (ite+1)==1 or (ite+1)%100==0:
            print("iteration: " + str(ite+1))
            gapBest = (self.best.cost-OPT)/OPT
            print("gapBest: " + str(gapBest))
            print("without Imp: " + str(ite-lastImp))
            curTime = time.time()
            print("CPU Time: " + str(curTime-startTime))

        if self.best.cost!=lastBest:
            lastBest = self.best.cost
            lastImp = ite
```

8.3 Result

Table 1: Execution i for instance Uruguay

iteration	gap_best	iterations without improvement	CPU time
1	0.0938527719493	0	27.89137077331543
100	0.0864044669717	97	2759.617785215378
200	0.0864044669717	197	5524.273468494415
300	0.0864044669717	297	8299.034429311752
400	0.0864044669717	397	11075.615463495255
500	0.0864044669717	497	13842.467777252197
600	0.0864044669717	597	16590.45319223404
700	0.0864044669717	697	19366.442168951035
800	0.0864044669717	797	22116.57707977295
900	0.0864044669717	897	24893.0479323864
1000	0.0864044669717	997	27642.713374614716

Table 2: Execution i for instance Canada

iteration	gap_best	iterations without improvement	CPU time
1	0.101795684556	0	5420.041585683823

Table 3: Summarization

Instance	Best gap	Worst gap	Average gap	Average time
Canada	0.101795684556	0.101795684556	0.101795684556	5420.041585683823
Uruguay	0.0864044669717	0.0938527719493	0.0870815856	2512.97394315

Table 4: CPU time

Iteration	CPU time (from begining)	CPU time (for each iteration)
1	27.89137077331543	27.89137077331543
100	2759.617785215378	2731.72641444
200	5524.273468494415	2764.65568328
300	8299.034429311752	2774.76096082
400	11075.615463495255	2776.58103418
500	13842.467777252197	2766.85231376
600	16590.45319223404	2747.98541498
700	19366.442168951035	2775.98897672
800	22116.57707977295	2750.13491082
900	24893.0479323864	2776.47085261
1000	27642.713374614716	2749.66544223

8.4 Explanation

Due to we only tested the first iteration of Canada instance, so on the Table 3, we copied the result of first iteration to the summarization. For the Uruguay instance, from the Table 1, we got that the worst gap and the best gap for the summarization and we computed mean value of the gap_best from Table 1, consisting with 11 set of results. The average time for Uruguay instance in summarization is computed by the Table 4: calculated the mean value of the column CPU time(for each iteration). If we didn't consider the first iteration CPU time(because it has a big difference with the other iteration), the Average time for Table 3 will be 2761.48220038.