Question 1

1. For my simple TS (`q1_simple.py`), I encoded the problem such that each solution is a permutation of numbers 1 to 20 stored in a list. I defined the neighborhood as all solutions that can be created by swapping 2 numbers of the current solution. I implemented the move operator as a function that sorts all neighbors in ascending order by cost and returns the first (lowest cost), admissible neighbor. I set the tabu list size (and the tabu tenure) as 5. I selected 100 iterations as the stopping criterion.

2. Running TS with `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]` as the initial solution returns:

```
[4, 2, 18, 17, 3, 19, 15, 14, 5, 10, 11, 8, 12, 7, 1, 16, 20, 9, 13, 6]
cost: 1315
```

3. Running TS with 10 different starting points (`q1_10_initial_solutions.py`) returns:

```
1 : ============
initial solution: [13, 14, 2, 9, 16, 7, 20, 5, 8, 6, 10, 4, 3, 12, 18, 1, 15, 17, 19, 11]
cost: 1701
best solution: [17, 8, 11, 4, 16, 13, 20, 15, 19, 9, 5, 7, 12, 2, 14, 6, 1, 10, 18, 3]
cost: 1294
2 : ============
initial solution: [5, 19, 3, 9, 4, 8, 13, 15, 14, 11, 7, 16, 2, 17, 1, 10, 20, 18, 6, 12]
cost: 1769
best solution: [6, 1, 10, 3, 9, 13, 7, 12, 18, 14, 5, 20, 15, 2, 19, 17, 8, 11, 4, 16]
cost: 1297
3 : ============
initial solution: [2, 3, 19, 12, 6, 17, 13, 11, 5, 15, 10, 4, 1, 14, 16, 20, 9, 18, 7, 8]
cost: 1834
best solution: [3, 6, 20, 13, 9, 1, 7, 15, 8, 12, 18, 10, 2, 11, 14, 17, 5, 19, 4, 16]
cost: 1314
4 : ============
initial solution: [8, 19, 18, 5, 12, 15, 10, 20, 11, 14, 2, 1, 13, 3, 17, 16, 6, 7, 4, 9]
cost: 1671
best solution: [13, 5, 19, 4, 17, 20, 8, 15, 2, 18, 7, 12, 11, 14, 16, 6, 1, 10, 3, 9]
cost: 1318
5 : ============
```

```
initial solution: [8, 10, 4, 13, 16, 3, 2, 14, 1, 7, 9, 5, 12, 17, 19, 6,
15, 11, 20, 18]
cost: 1771
best solution: [3, 14, 12, 16, 9, 18, 2, 4, 11, 17, 10, 19, 7, 1, 5, 6,
15, 20, 8, 13]
cost: 1336
6 : ============
initial solution: [20, 9, 12, 17, 1, 14, 8, 13, 4, 11, 16, 3, 2, 19, 18,
15, 6, 7, 10, 5]
cost: 1665
best solution: [3, 18, 19, 4, 17, 10, 14, 2, 11, 16, 9, 12, 15, 8, 1, 6,
13, 20, 7, 5]
cost: 1307
7 : ============
initial solution: [19, 3, 16, 9, 2, 1, 20, 11, 10, 8, 6, 12, 15, 14, 4,
18, 17, 7, 13, 5]
cost: 1694
best solution: [17, 4, 11, 18, 16, 5, 20, 8, 15, 19, 13, 7, 12, 2, 14, 6,
1, 10, 3, 9]
cost: 1298
8 : ============
initial solution: [11, 5, 13, 2, 3, 14, 9, 17, 6, 10, 1, 15, 4, 20, 7, 8,
19, 12, 16, 18]
cost: 1761
best solution: [3, 10, 1, 7, 6, 9, 12, 8, 20, 13, 14, 2, 15, 19, 5, 16,
18, 11, 4, 17]
cost: 1300
9 : ============
initial solution: [8, 12, 13, 5, 7, 19, 1, 2, 3, 4, 9, 11, 16, 6, 14, 10,
17, 18, 15, 20]
cost: 1778
best solution: [3, 1, 7, 6, 13, 9, 10, 12, 20, 5, 18, 14, 2, 15, 19, 16,
11, 4, 8, 17]
cost: 1311
10 : ============
initial solution: [15, 12, 9, 5, 6, 14, 11, 1, 16, 18, 8, 2, 19, 17, 7,
13, 4, 10, 20, 3]
cost: 1722
best solution: [17, 7, 5, 1, 6, 4, 20, 8, 11, 13, 19, 15, 12, 10, 16, 18,
2, 14, 3, 9]
cost: 1287
```

Starting the search at 10 different initial solutions resulted in solutions with around 1300 in cost, the lowest being 1287 and the highest being 1336. These results are similar to the previous result with a cost of 1315. Therefore, the initial solution has little impact on the result of TS.

Changing tabu list size to a smaller and larger number (q1_smaller_larger_tabu_size.py) returns:

```
tabu list size = 1
[17, 1, 18, 10, 3, 4, 7, 20, 5, 6, 11, 8, 15, 19, 13, 16, 12, 2, 14, 9]
cost: 1310
tabu list size = 25
[6, 1, 10, 3, 9, 13, 7, 12, 14, 18, 5, 20, 15, 2, 19, 17, 8, 11, 4, 16]
cost: 1301
```

Changing the tabu list size from 5 to 1 resulted in a solution with a slightly lower cost (1310). Changing the size to 25 resulted in an even lower cost (1301). Within a reasonable range (not too small or large), there isn't a strong correlation between tabu list size and the quality of the best solution found by TS.

Changing the tabu size to a dynamic one (q1_dynamic_tabu_size.py) returns across 3 trials:

```
tabu list size: 2
tabu list size: 26
tabu list size: 24
tabu list size: 16
tabu list size: 13
[17, 1, 5, 18, 3, 4, 7, 20, 10, 6, 11, 8, 15, 2, 13, 16, 12, 19, 14, 9]
cost: 1312

tabu list size: 12
tabu list size: 11
tabu list size: 6
tabu list size: 18
tabu list size: 13
[17, 5, 13, 6, 9, 1, 7, 20, 10, 3, 11, 8, 15, 12, 18, 16, 4, 19, 2, 14]
cost: 1317

tabu list size: 25
tabu list size: 5
tabu list size: 14
tabu list size: 27
tabu list size: 5
[9, 3, 18, 6, 13, 14, 10, 2, 15, 19, 12, 1, 7, 20, 5, 16, 11, 4, 8, 17]
cost: 1316
```

I chose to range the tabu list size from 1 to 30 and changed it every 20 iterations (there are 100 iterations in total). This didn't make a big difference in the quality of the best solutions found by TS.

Adding an aspiration criterion for the best solution so far (q1_aspiration_global_best.py) returns:

```
[17, 1, 5, 18, 3, 4, 7, 20, 10, 6, 11, 8, 15, 2, 13, 16, 12, 19, 14, 9]
cost: 1312
```

Adding this aspiration criterion results in a solution with a slightly lower cost. The majority of departments were placed in a different location compared to the result of simple TS.

Adding an aspiration criterion for the best solution in the neighborhood (`q1_aspiration_local_best.py`) returns:

```
[17, 1, 5, 18, 3, 4, 7, 20, 10, 6, 11, 8, 15, 2, 13, 16, 12, 19, 14, 9]
cost: 1312
```

This is the exact same solution as TS with the previous aspiration criterion. Although the different aspirations could have resulted in different solutions, they didn't.

Using a neighborhood of size 20 (`q1_smaller_neighborhood.py`) returns:

```
[17, 1, 5, 6, 13, 3, 7, 8, 12, 9, 18, 15, 20, 14, 10, 4, 2, 19, 11, 16]
cost: 1349
```

This lowered the quality of the solution since there are fewer solutions considered at every iteration. So the best solution out of the previous 190 (at each iteration) must be one of the 20 in order to be considered. Lowering the neighborhood size trades quality for speed, since it's quicker to generate and sort 20 neighbors compared to 190.

Adding a frequency-based tabu list (`q1_frequency_tabu.py`) returns across 3 trials:

```
[6, 1, 10, 3, 9, 13, 7, 12, 18, 14, 5, 20, 15, 2, 19, 17, 8, 11, 4, 16]
cost: 1297

[13, 5, 17, 19, 4, 6, 7, 20, 15, 2, 10, 1, 8, 12, 11, 9, 3, 18, 14, 16]
cost: 1313

[3, 10, 18, 19, 6, 9, 14, 15, 2, 13, 1, 12, 7, 20, 5, 16, 11, 8, 4, 17]
cost: 1307
```

I used Restart Diversification by storing visited solutions in a set. I convert each solution from a list of numbers to a string of numbers separated by commas. Every 20 iterations, I restart the search at an unvisited solution using the frequency-based tabu. This resulted in a better solution than simple TS on average. Simple TS likely wasted iterations after reaching its best solution. This version of TS reduces the number of wasted iterations by restarting the search in a new region of the search space every 20 iterations, meaning the maximum number of wasted iterations in one area is 19.

Question 2

a) Representation of the solution:

K_p ranges from 2 to 18 and has a precision of 2 decimal points. This means that there are 1601 different possible values. $Log_2 1601 = 10.65$ so 11 binary digits are needed to represent K_p. 00000000000 represents 2.00, 00000000001 represents 2.01, and so on.

T_I ranges from 1.05 to 9.42, meaning there are 838 different possible values. $Log_2 838 = 9.71$ so 10 binary digits are needed. 0000000000 represents 1.05, 0000000001 represents 1.06, and so on.

T_D ranges from 0.26 and 2.37, meaning there are 212 different possible values. $Log_2 212 = 7.73$ so 8 binary digits are needed. 00000000 represents 0.26, 00000001 represents 0.27, and so on.

Question 3

Evaporation Rate = 10:

	Population = 30	**Population = 50**	**Population = 100**
Diffusion Rate = 40	9012 ticks	6046 ticks	1191 ticks
Diffusion Rate = 80	7710 ticks	6046 ticks	1517 ticks

Evaporation Rate = 20:

	Population = 30	**Population = 50**	**Population = 100**
Diffusion Rate = 40	7207 ticks	5037 ticks	3705 ticks
Diffusion Rate = 80	7106 ticks	6130 ticks	4051 ticks

As the population size increased, the colony found all the food faster. This makes sense because even if the effectiveness of each ant remains constant, increasing the total number of ants covers more ground (search space) and results in finding all the food faster. Once food is found, more ants will follow the pheromone trail and deplete the entire piece of food faster as well.

Increasing the diffusion rate doesn't always improve performance. When the population size was 30, increasing the diffusion rate improve the performance. However, when the population size was 50, increasing the diffusion rate didn't make a difference. When the population size was 50, increasing the diffusion rate actually lowered performance.

Increasing diffusion rate widens the pheromone trail. This makes it easier for other ants to find the trail. However, ants that found the pheromone trail has a more difficult (and wider) trail to follow. When the population size is small, getting ants to discover trails is a bigger challenge, so increasing the diffusion rate is beneficial. When the population size is large, ants are able to find trails simply because there are more ants. Increasing the diffusion rate lowers the effectiveness of ants that already found the trail, which lowers overall performance.

Increasing the evaporation rate doesn't always improve performance. When the population size is 100, increasing the evaporation rate worsens performance, likely because a higher evaporation rate means trails last a shorter amount of time and lead fewer ants to the food. An ant that would've reached food by following a trail may not reach the food if the trail evaporates too quickly and before the ant arrives at food.

When the population size is 30, increasing the evaporation rate improved performance. After an ant finds the last piece of food in an area, they still leave a trail. Ants that follow that trail can be led to nothing. Having trails evaporate quicker can reduce the probability of such a situation.