1. Fast Genetic algorithm with Clustered greedy initialization

20170620 SeongwoongJo

Before start, I've never seen any other papers and references except for the slides from the lecture and 'pmx' algorithm. All the implementations and experiments are made from me

2. Introduction

2.1 Key Idea

1. Parallel GA for python multiprocessing pool
2. Good parents make a good offspring. Let's focus on the initialization - Kmeans clustring + greedy algorithm
3. Large-scale experiments for testing

2.2 Example usage and description

If you just give population and fitness limit, other hyperparameters are automatically designated.

```
python solver.py -t rl11849.tsp -p ${population} -f ${fitness_limit}
```

else, if you want to control other parameters rather than population and fitness limit use the follow command.

```
python solver.py -t rl11849.tsp -p 500 -f 100000000 -n 20 -g 500 \
--elitism_rate 0.2 --init partial_greedy --crossover my \
--greedy_ratio 1 --num_clusters 16 --kmeans_iter 500 --save_dir ./logs/results

-t : the location of .tsp file to load
-p : number of population
-f : maximum fitness function call
-g : generation
-n : number of multiprocessing worker

--elitism_rate : elitism_rate for the maximum population
--init : the mode of initialization
--crossover : the mode of crossover

--greedy_ratio : Percentage of cities to apply the greedy algorithm
--num_clusters : initial number of the clusters
```

```
--kmeans_iter : k-means clustering iterations
--save_dir : where to save logs and hyperpameter informations
```

2.3 Observation and Motivation

I think that it is very important to control the balance between randomness(diversity) and superiortiy of each population. For the fixed population size, diversity and superiority are a relationship of trade-off. For example, as we search for the large space, the points are getting sparse, which means that diversity is increasing and superiority of each chromosome is decreasing. So, I focus on the initialization method which makes the superior initialized population. Specifically, My method is reducing search space using k-means clustering and making great parents using (partial) greedy algorithm. In the term (partial) greedy algorithm is conducted by simply doing greedy algorithm on the subset of the whole city, and subset size is controlled by greedy_ratio. (greety_ratio 0 for the random initializaiton and 1 for the original greedy algorithm) Finally, I can control the balance by greedy_ratio and cluster numbers. This problem(rl11849) has >10000 cities, > 10000! search space. Thus, clustering effectively reduces the search space and execution time.

3. Algorithm and Implementation

3.1 Parallel Genetic Algorithm

I implement multiprocessing genetic algorithm using python 'multiprocessing' library. I can boost the training speed by using 20 workers simultaneously. Below is Example usage of multiprocessing

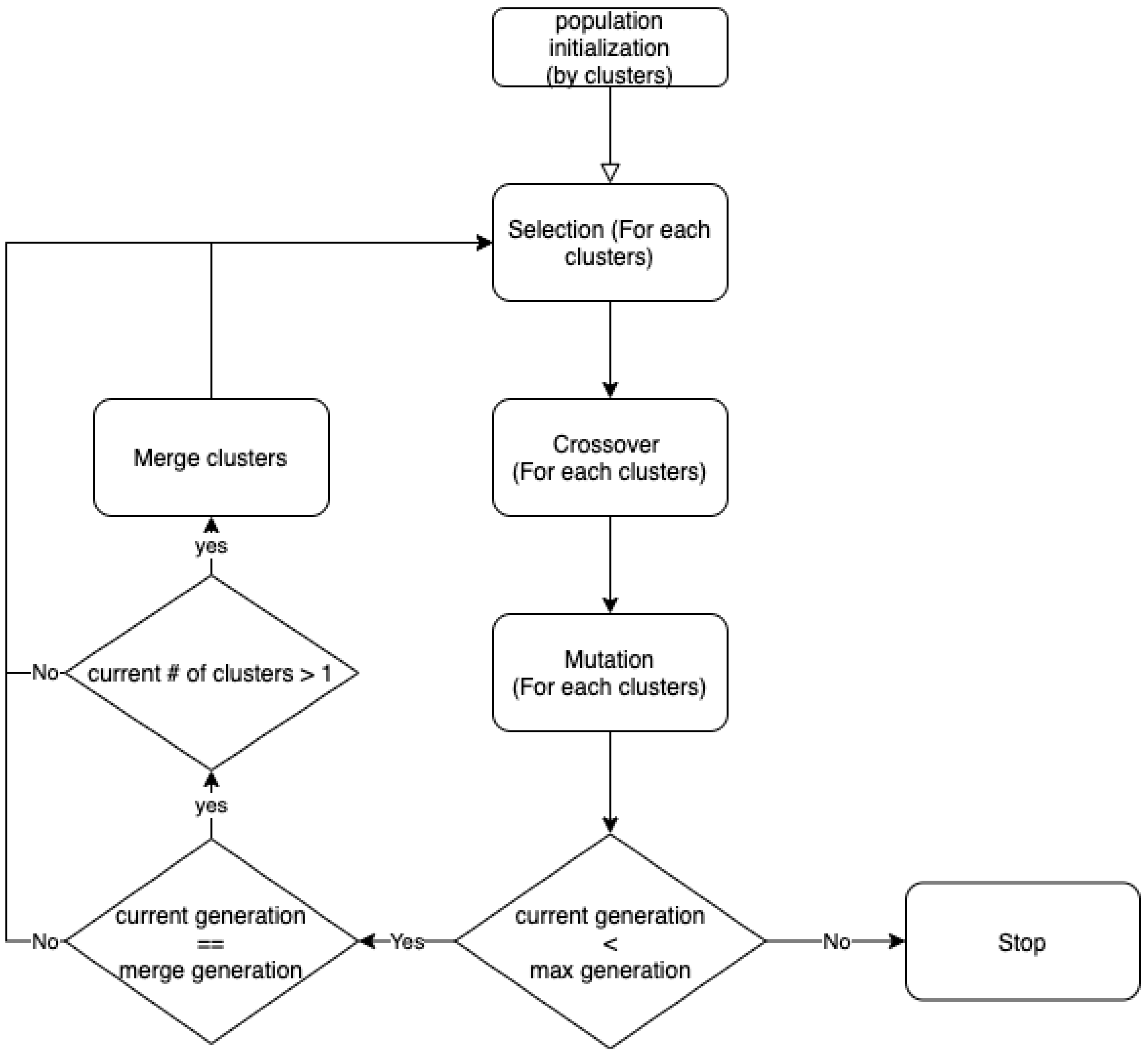
```
from multiprocessing import Pool

def func(arg):
  return arg

pool = Pool(num_workers)
## args is list of arg
l = pool.map(func, args)
```

I adjust this procedure on all the genetic procedure: initialization, selection, crossover, mutation

3.2 Algorithm

3.2.1 Block Diagram of algorithm

The main difference between my algorithm and normal GA is that my algorithm has additional k-mean clustering block and cluster Merging block.


population initialization (by clusters)
Selection (For each clusters)
Crossover (For each clusters)
Mutation (For each clusters)
Merge clusters
yes
current # of clusters > 1
No
yes
current generation == merge generation
No
Yes
current generation < max generation
No
Stop

Below is the psuedo process

```
Input : K(num_clusters), pop_size(population_size)

1. Divide all cities into K group(cluster), and initialize each group by the partial g
2. For each group, Do GA(selection, mutation, procedure)
  2-1. Merge two group into single group when satisfying the merging condition.
       After merging, the number of group should be half and each group's population s
3. Repeat 2 until the end condition
```

3.3.2 Step by Step code analysis

Selection, Mutation, Crossover is same as the general procedure. Thus in this part, I will explain the three components simply.

3.3.2.1 Initialization

- K-means clustering

```
def k_means_clustering(datas, num_clusters = 1, n_iter = 1000):
    xs = []
    ys = []
    for x,y in datas:
        xs.append(x)
        ys.append(y)

    xmin, xmax = np.min(xs), np.max(xs)
    ymin, ymax = np.min(ys), np.max(ys)

    centroids = []
    for _ in range(num_clusters):
        centroids.append((np.random.uniform(xmin,xmax),np.random.uniform(ymin,ymax)))

    for _ in tqdm(range(n_iter)):
        ## update points
        base_sets = [[] for _ in range(num_clusters)]
        for o,(ptx,pty) in enumerate(datas):
            min_dis = np.inf
            loc = 0
            for i, (cent_x, cent_y) in enumerate(centroids):
                dis = ((ptx-cent_x)**2 + (pty - cent_y)**2)**0.5
                if dis < min_dis:
                    min_dis = dis
                    loc = i
            base_sets[loc].append(o)

        ## update centroids
        for i in range(num_clusters):
            x_cent = 0
            y_cent = 0
            for o in base_sets[i]:
                x_cent += datas[o][0]/len(base_sets[i])
                y_cent += datas[o][1]/len(base_sets[i])
            centroids[i] = (x_cent,y_cent)
    return base_sets, centroids
```

For given datas, conduct kmeans-clustering with the given num_clusters and iterations. As you can see in (1), initial centroids are initialized by (U[Lx,Hx],U[Ly,Hy]) where U is uniform distribution and each of Lx,Ly is the lowest value of x,y among all datas, and Hx,Hy is the highest value of x,y among all datas. Finally, the above function returns centroids and the list of base_set, which is a set of cities corresponding to each centroid.

- Partial greedy algorithm

```
def partial_greedy(args):
    """
    주어진 도시들에서 greedy_ratio의 비율만큼 random subset을 뽑고 해당 지역들에 대하여 greedy algor
    """

    datas, greedy_ratio,base_set = args
    random.shuffle(base_set)
    pivot = max(1,int((1-greedy_ratio) * len(base_set)))
    chromosome = base_set[:pivot].copy()
    new_base_set = base_set[pivot:].copy()
    while len(new_base_set) > 0:
        current_node = chromosome[-1]
        min_dist = np.inf
        min_query_arg = 0
        for arg,query_node in enumerate(new_base_set):
            dist = calc_distance(datas[current_node], datas[query_node])
            if dist < min_dist:
                min_dist = dist
                min_query_arg = arg
        chromosome.append(new_base_set.pop(min_query_arg))

    fit = fitness(chromosome,datas)

    return chromosome, fit
```

Partial Greedy Algorithm is a simple extension of the greedy algorithm and is able to control the randomness. For each base_set in base_sets, partial greedy algorithm is conducted with the given greedy_ratio and population. The above code is a original version of the partial greedy algorithm which only consider 1-way greedy algorithm.

3.3.2.2 GA procedure(Selection, Mutation, Crossover)

The GA procedure is done on each of the base_set.

- Selection

```
def elitism_selection(population, elitism_rate):
    """
    population : List of (chromosome, fit)
    elitism_rate :

    return (chromosome, k)
    """
    elitism_k = int(len(population)*elitism_rate)
    tournament_pool = random.sample(population, elitism_k)
    result = sorted(tournament_pool, key=lambda x: x[1], reverse=False)
    return result[0]
```

I adopted 'elitism selection' for the whole process. elitism_k is calculated by the product of elitism_rate and population size.

- Mutation

```
def mutate(sequence, datas):
    new_sequence = sequence.copy()
    i,j = random.sample(range(len(sequence)),2)
    if i > j:
        i,j = j,i
    prob = np.random.uniform()
    if  prob< 0.05:
        ## random window switching
        new_sequence = new_sequence[:i] + new_sequence[j:] + new_sequence[i:j]
    elif prob < 0.1:
        ## random two city switching
        new_sequence[i],new_sequence[j] = new_sequence[j],new_sequence[i]
    fit = fitness(new_sequence,datas)
    return new_sequence, fit
```

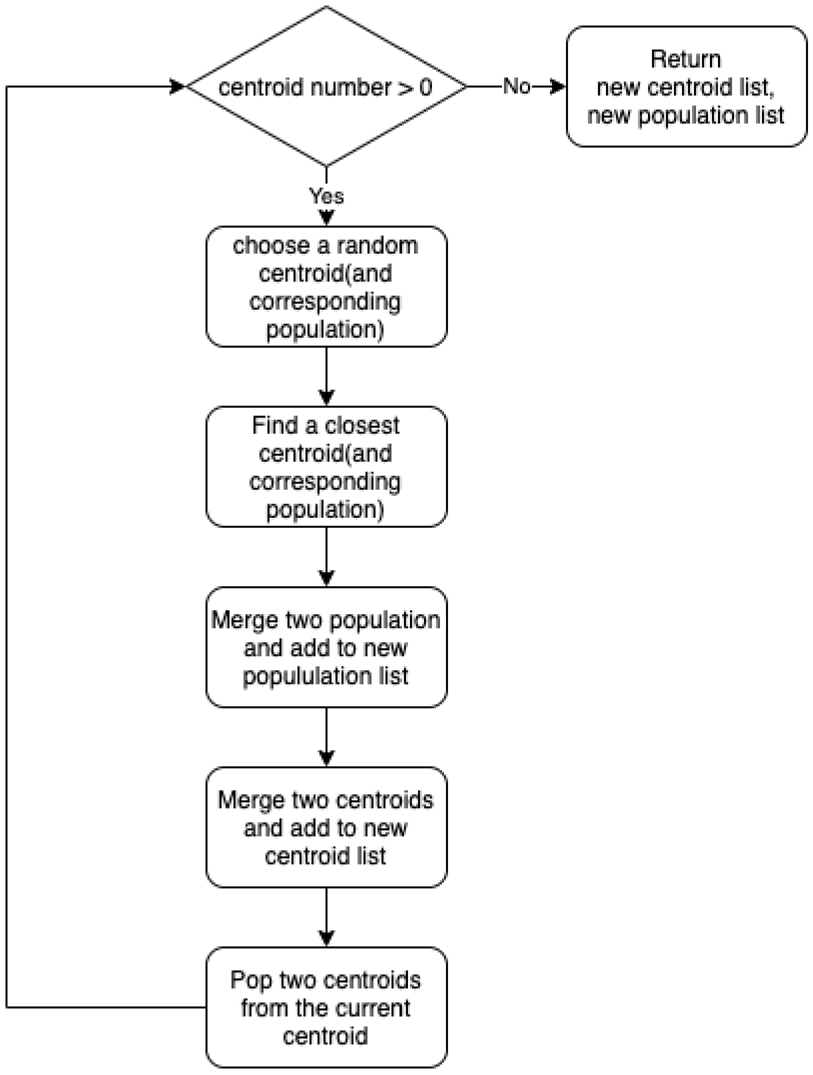
I adopt two mutation policies. One is random window switching, and the other is random two city switching. Each of the mutation is performed with the probability 5%

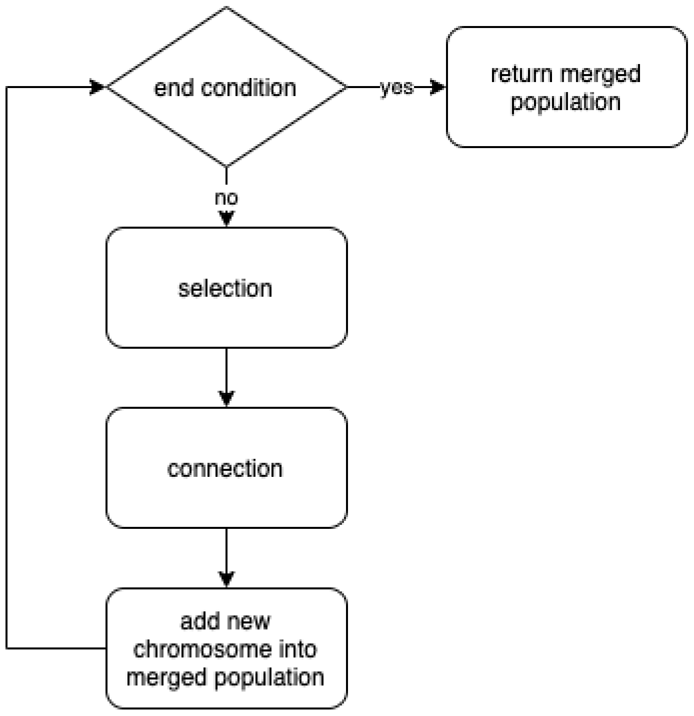
- Crossover

```
def my(p1,p2):
    if np.random.uniform()<0.5:
        return pmx(p1,p2)
    else:
        return order(p1,p2)
```

To increase the diversity of the population, I combine two different Crossover policies. The first one is pmx crossover, which is usually used in tsp problem. And the other is order crossover, which is explained in the professor's lecture note. Each of the crossover is perforemd with the probability 50%.

3.3.2.3 Merge clusters


centroid number > 0
No
Return new centroid list, new population list
Yes
choose a random centroid(and corresponding population)
Find a closest centroid(and corresponding population)
Merge two population and add to new popululation list
Merge two centroids and add to new centroid list
Pop two centroids from the current centroid



```python
def connect(chromosome1,chromosome2):
    i = random.randint(0,len(chromosome1)-1)

    new_chromosome = chromosome1[:i] + chromosome2 + chromosome1[i:]
    return new_chromosome
```

The first diagram is overall process of merging clusters, and the second diagram is merging population policy that I used and suggested. There are many policies can be applied to population merging. In my case, I use the below algorithm, which is motivated from genetic algorithm's selection stage. If you see the below algorithm, you can know merging population is similar to GA except for removing mutation and replace crossover with connect. Selection and Connect algorithm's purpose is to make good connectivity from the superior parents. (i.e a kind of selection pressure)

4. Experimental results

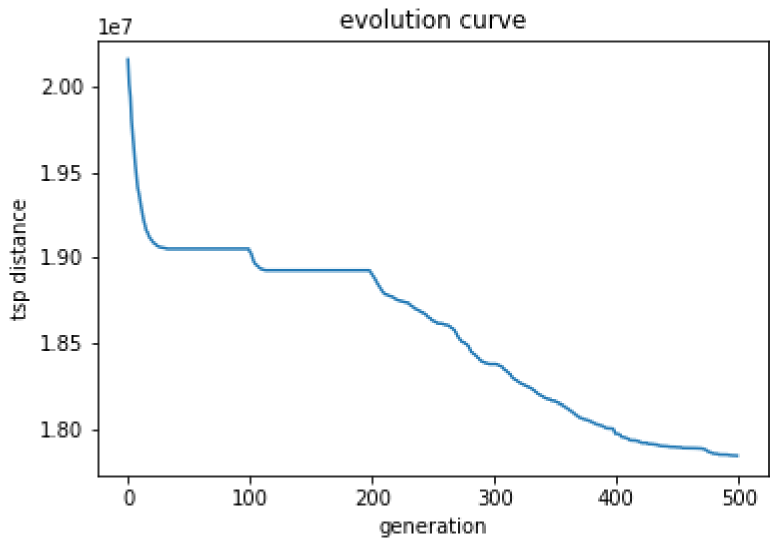
The evaluation of the algorithm is viewed on two perspective: 1.Time and 2.Performance

4.1 Parallel GA

num workers	time
1	836.949
20	59.723

The time is calculated on the condition of greedy initialization(when greedy_ratio = 1) with population_size = 20 We can observe that parallel GA with many workers can speedup the time tremendously.

4.2 Evolution Curve



The above curve is the example curve when num_clusters = 16, merge_g = 100. On the every beginning iteration right after the merge, the distance rapidly decreases. You can observe that on every iteration around multiples of merge_g(=100), there is sudden improvement. We can explain these phenomenons through the size of search space. When num_clusters is high before merge, each of clusters has small amount of cities and small search space. So, it can converge rapidly. On every merge step, it gradually increase possible search space and escape from the local minima.

4.3 Clustered greedy algorithm

4.3.1 Default setting

I run totally 25 experiments for the comparison. The final value can be changed according to the seed, but an approximate trend might be right.

- Running Experiments

```
./run_exps.sh
```

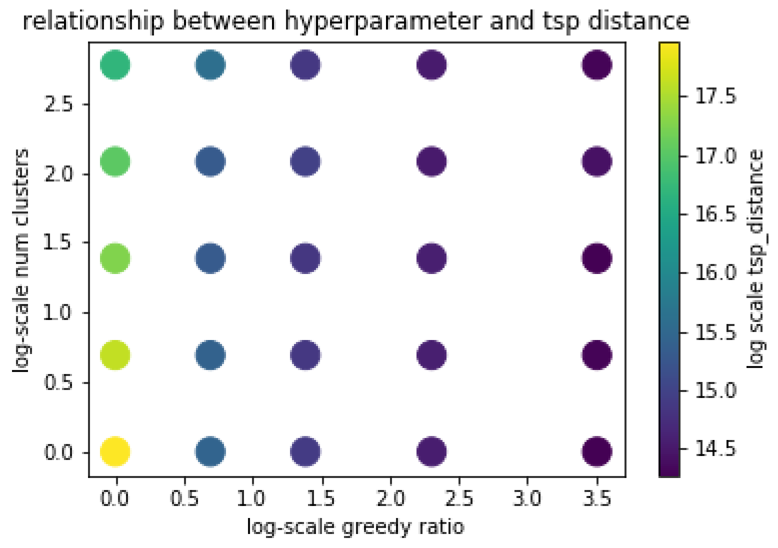
- Fixed Hyperparameters

```
num_workers : 20
population : 500
generation : 500
fitness_limit : 100000000 ## no limitness of fitness call
elitism_rate : 0.2
init : partial_greedy
crossover : my
kmeans_iter : 200
merge step : linearly scheduling (= max_generation//(np.log2(num_clusters)+1))
```

- Search Space

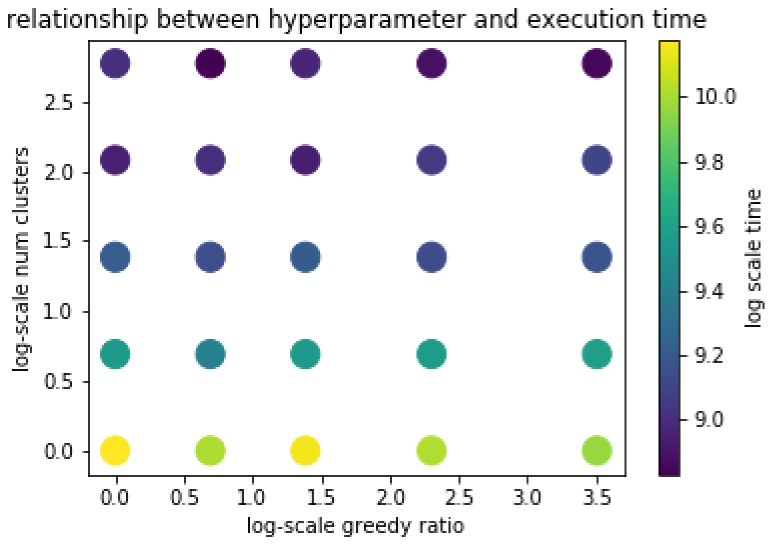
```
greedy_ratio : [0 0.5 0.75 0.9 0.97]
num_clusters : [1 2 4 8 16]
```

4.3.2 Hyperparameters-Performance result



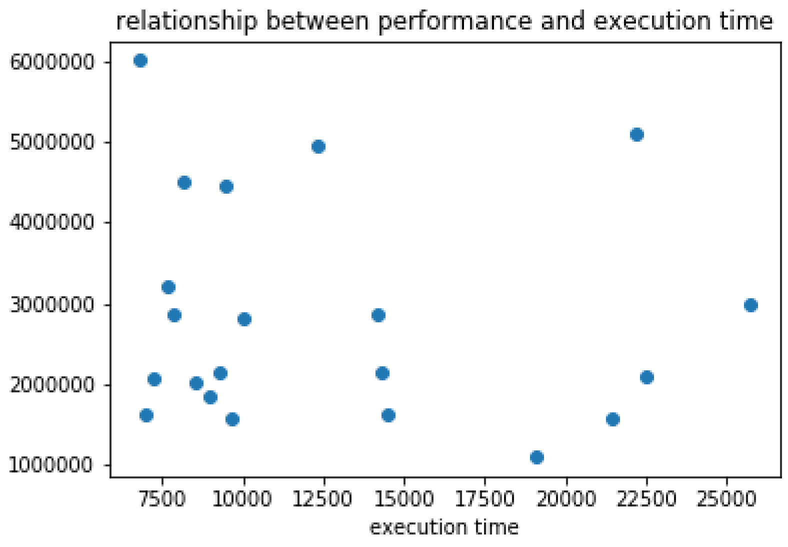
When the greedy ratio is low(i.e initialization is not effective), you can see that many num clusters effectively reduce the search space and makes good intialization effect. However, when greedy ratio goes up, too many num clusters badly affects to the performance. I guess that this is because greedy algorithm gives very good approximation to the tsp.(and k-means clustering disrupts that procedure)

4.3.3 Hyperparameters-Time result



As the greedy ratio is lower and the number of clusters is higher, the time is reduced more. Since greedy ratio only affects to the duration of initialization, the number of clusters will much more affects to the time as the max GA generation is higher.

4.3.4 Time-Performance result



	distance	GA time	init time
Best-time exp	1619243.71	6987.94	22.905
Best-distance exp	1100685	19100.5	1344.6

By comparing two points, one is best-time experiment and the other is best-distance experiment, the best-time experiment gains >270% time benefit with <50% distance loss. On the restrict computing resource, K-means clustering with large K will be helpful by giving large time-benefit.

4.4 My Final Submission

My final submission result is 1,094,827.02 and achieved by the below hyperparameter options. Since the purpose is achieving high score, I set population as a very large number(12000). With >1 num clusters and high greey ratio and low generation, I can catch both time and performance.

```
"num_workers": 20,
"population": 12000,
"generation": 20,
"elitism_rate": 0.4,
"init": "partial_greedy",
"crossover": "my",
"greedy_ratio": 1.0,
"num_clusters": 4
```

5. Failed approaches

5.1 mcmc initialization

I tried mcmc initialization with expecting effective search, but it fails because of the too-large search space and too-many fitness call. The number of fitness call is equals to (sampling iteration * population size). Since the space is too large, sampling iteration also goes large, then the final number of fitness call explodes.

5.2 stochastic greedy initialization

I refered to the professor's comment from the campuswire. He suggested N-way greedy algorithm and I designed stochastic greedy algorithm by mixing 1-way and 2-way greedy algorithm. The main advance is randomly choose between 1-way greedy ,2-way greedy on the each step of greedy algorithm. (original greedy algorithm is just use 1-way greedy on the every step). I expected that it can greatly increase diversity while maintaining superiority. But, the results says that it is worse than original, even takes longer time.

6. Conclusion

I designed Clustered greedy initialization for many cities tsp problem. I suggests novel clustering and merging method that can be adjust on the GA, and It shows great speedup on the TSP. With totally random initialization, My Clustered greedy GA boosts both speed and performance. However, sadly, when the initialization method is very effective originally(like greedy algorithm with greedy_ratio = 1), the performance gets lower on the clustering methods. Also, I tried mcmc initialization and novel stochastic greedy initialization, but It did not give good result. However, though there's some flaws on my algorithm, I expect that it will give great help to those who don't have enough computing resources.