AI based Software Engineering Assignment #1

20170344 Minjae Song

Intro

At beginning, I used usual evolutionary computation. For usual selection, I used tournament selection. For crossover, I used sequence crossover. For generational selection, I used elitism.

My method's process is as follows.

<sudo code for evolutionary computation>

```
init population
loop for #generation{
        evaluate fitness
        crossover for children
        mutation
        select for next generation
}
```

I used python. My most significant element of testing was "time". There are several approaches for TSP problem (e.g. ACO, PSO…). But, I think using ACO or PSO needs reasonable population size and generation number to see some improvements. Because ACO and PSO have merits in aspect of interaction between each individual and, with lower population size and generation number which is my case, I thought they don't have such thing to convey between individuals.

Result

Test for optimization and finding hyper parameter at first.

For understanding, I placed each parameter value and short description.

<Trial #1>
Population size = 30
Generation size = 100
Mutation rate = 0.2
Result = 86421898.99291812

Basic condition. Did not use elitism. Just put all children into next generation.

<Trial #2>
Population size = 100
Generation size = 10
Mutation rate = 0.2
Result = 86315003.07999954

By comparing #2 result with #1, I wanted to see the which parameter is more important between generation size and population size. Other than two parameter, all condition same as #1.

<Trial #3>
Population size = 10
Generation size = 200
Parent number = 2
Result = 86644643.55999997

With high generation size however, results in meaningless offspring.

<Trial #4>
Population size = 100
Generation size = 100
Parent number = 10
Elite number= 8
Mutation rate = 0.2
Result2 = 85949858.64000027

Brought in Elitism. From #3, it seems mutation does not bring offspring to development. So, I thought that bringing elite from parents generation may lead generation to better results. Actually, it improved very little.

<Trial #5> gradual replacement
Population size = 100
Generation size = 100
Parent number = 80
Elite number= 8
Mutation rate = 0.2
Result = 85630890.1094121

Used gradual replacement for generational selection. Thought to lead generation into proper (which brings more development) direction, not much children is needed.

Optimization

From several tests, I realized that just pushing generation into many loops or putting more population is like waiting for a miracle. Therefore, I decided to optimize in "init population" part.

Padding for initialized population

Using law of large number, I picked 100 individual and picked best of them. Then, they become one of the initialized population. By doing this, each init population is at least better than 100 different routes.

Local Search

For better initial population, I also used local search. Here comes fitness evaluation limit parameter for limiting local search.

Also, for finding local optima, which we want to find, after number of fitness evaluation limit, I used improvement evaluation. So, even we passed fitness evaluation limit, if local search is still in improvement, fitness evaluation is keep going on.

```
improve = (pre_fit - route_fit) / pre_fit
```

Crossover Method

To make evolutionary computation meaningful, I thought that effective crossover method was needed.

Several ways to execute crossover.

1. Partially Mapped Crossover(PMC, PMX)

This approach was based on randomly picking points. After randomly picking, slice parent's gene into three parts and replace one part with other parent's part of same slice. Then, placing empty parts using relationship earned by two parents' slices which was exchanged.

2. Cycle Crossover(CX)

Main idea of CX is to conserve one parent's cycle. By picking cycle in one's parent, remaining part will be covered by other parent's gene.

There are other Crossover methods. But, from my search, many of them are based on either PMC or CX. Plus, even pure CX costs a lot of time. CX made about 10minutes for one generation which is clearly heavier than PMC or normal crossover.

Improvement Result

<Trial #6>

Population size = 5

Fitness evaluation limit = 2000 (only for local search per one init individual)

Generation size = 10

Parent number = 2

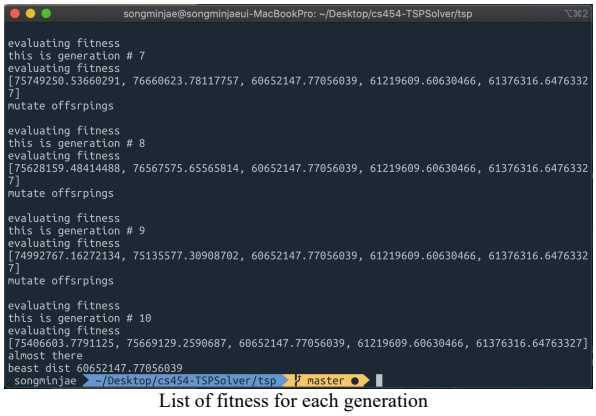
Elite number= 3

Mutation rate = 0.2

Neighbor size = 50

Result = 60652147.77056039

From now all test condition is having gradual replacement, padding for init population, local search with neighbors.



List of fitness for each generation

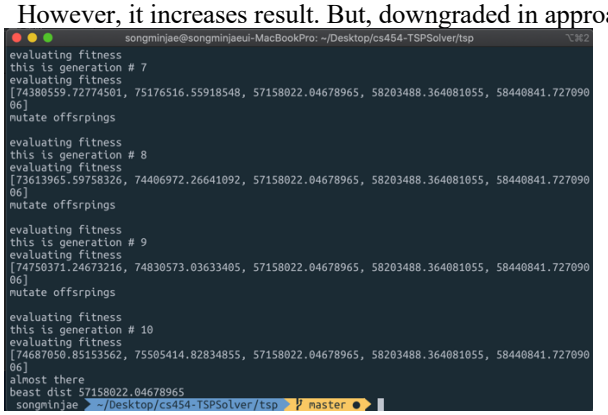
<Trial #7>

Population size = 5
Fitness evaluation limit = 2000 (only for local search per one init individual)
Generation size = 10
Parent number = 2
Elite number= 3
Mutation rate = 0.2
Neighbor size = 100
Result = 57158022.04678965

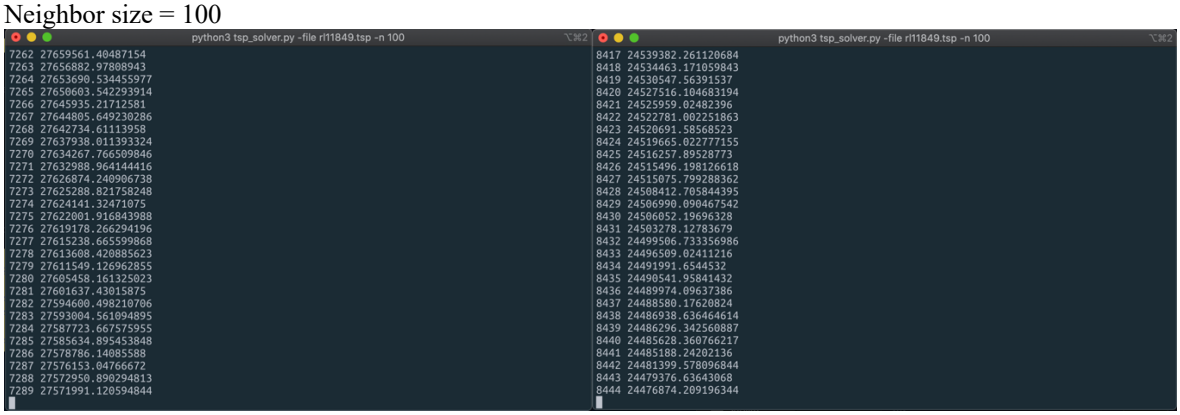
Increased neighbor size to 100. I couldn't handle variance of neighbors. So, I increased possibility of diversity by picking more neighbors. Meaning that randomly pick 2 points to tweak route for 100 times makes routes neighbors more diverse than 50 times.

This method can control diversity and makes control of neighbor size in probability side, but makes calculation time significantly increase and anyway it is still probability approach.

However, it increases result. But, downgraded in approach of time consumed.



<Trial #8>
Fitness evaluation limit = infinite (only for local search per one init individual)
Neighbor size = 100

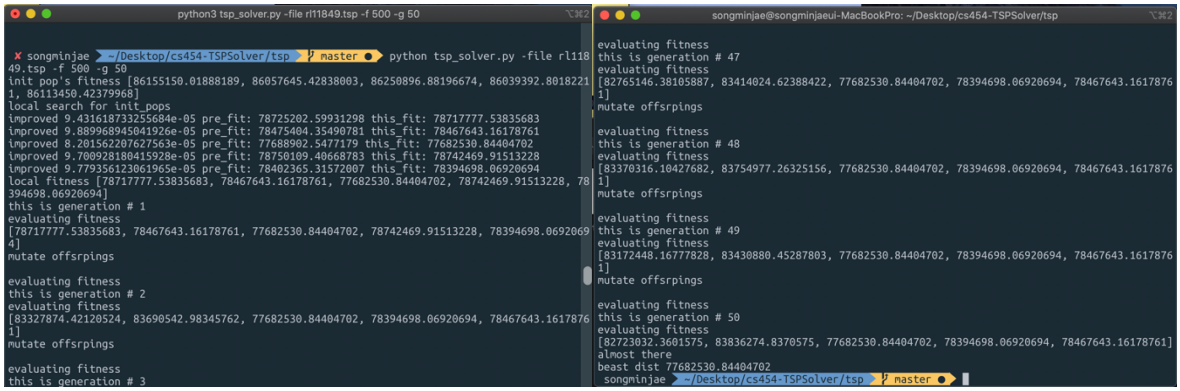


each row (number of evaluation in local search, evaluated distance)

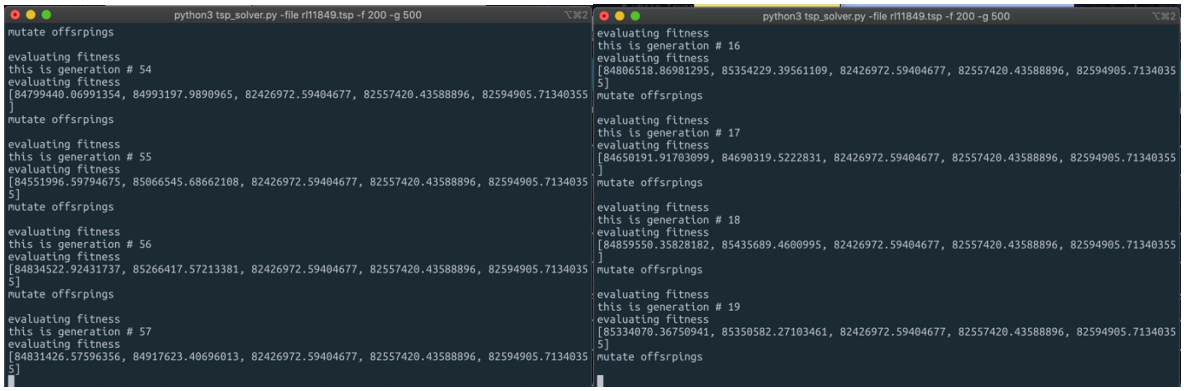
I wanted to find the limit of local search. To help this, I set fitness evaluation limit to infinity and monitored improvements.

The result was that when we approached about 8000th local search, it seems improvement came close to limit. The improvement of distance was about thousand units scale. This is significantly lower improvement compared to thousand -hundred thousand units scale improvement under 2000th local search.

<Trial #9>
Population size = 5
Fitness evaluation limit = 500 (only for local search per one init individual)
Generation size = 50
Parent number = 2
Elite number= 3
Mutation rate = 0.2
Neighbor size = 50
Result = 77682530.84404702



Increase of generation size doesn't make much improvement.



But as you can see from above images, every evolution loop has fixed three elements which is from parent part.

We should fix the mutation, crossover algorithm for competitive child.

The problem is that, fitness that was earned using local search stick inside during evolutionary loop.

I firstly thought the reason is that children are not as competitive as parent.

To make children more competitive, there maybe 2 ways.

First is to increase generation number so that brilliant mutation or crossover make surprising improvement.

=> fix generation size

Second is to reduce mutation rate. Previous mutation rate is 0.2.

This means 0.2*11894, almost 2379 cities change during improvements. This may be too much.

=> fix mutation rate

<Trial #11>

Population size = 5

Fitness evaluation limit = 200 (only for local search per one init individual) for time saving… I should have to reduce this parameter ☹

Generation size = 500

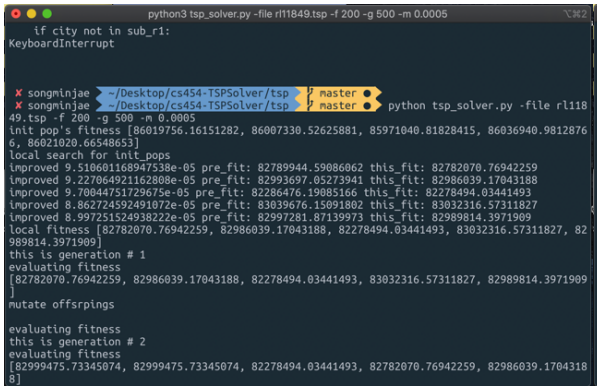
Parent number = 2

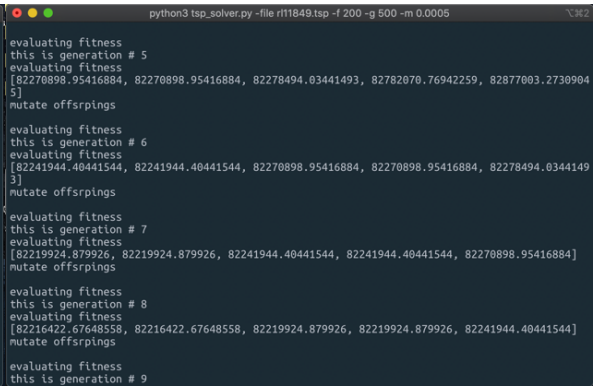
Elite number= 3

Mutation rate = 0.0005

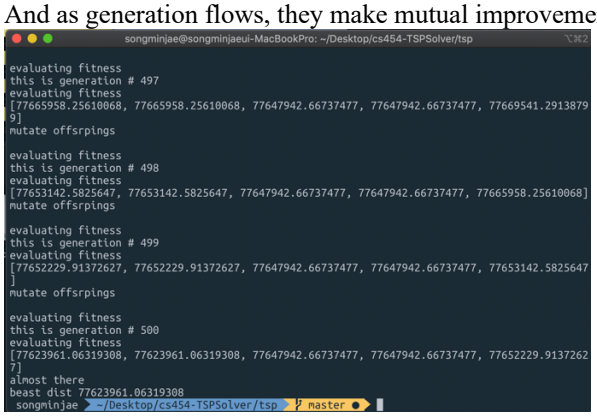
Neighbor size = 50

Result = 77623961.06319308

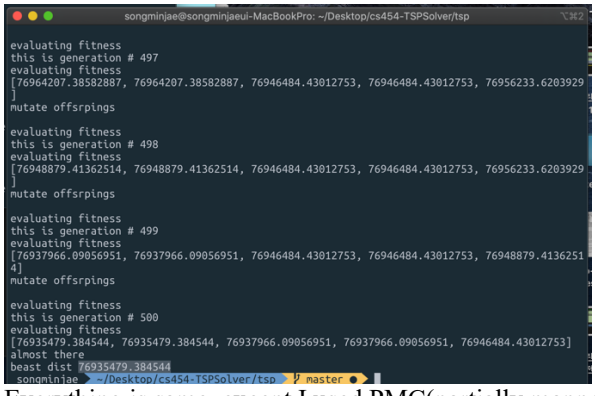




This makes evolutionary child better competitiveness.
And as generation flows, they make mutual improvements.



<Trial #11>
Population size = 5
Fitness evaluation limit = 200 (only for local search per one init individual)
Generation size = 500
Parent number = 2
Elite number= 3
Mutation rate = 0.0005
Neighbor size = 50
Result = 76935479.384544

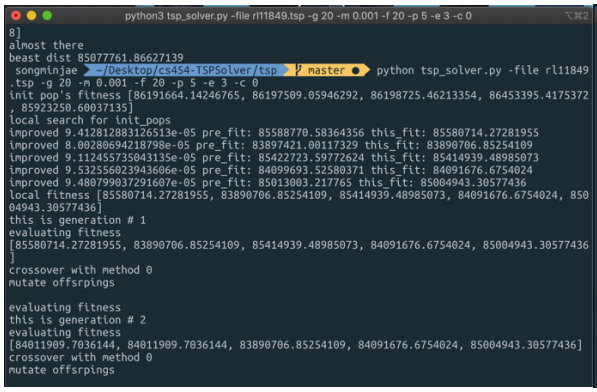


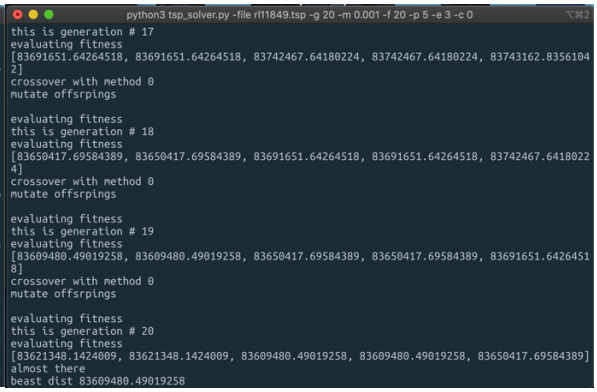
Everything is same, except I used PMC(partially mapped crossover)
But, there exists pair inside same generation, due to reduction of mutation rate, not much diversity exists. Therefore, many best_child(which is best picked in crossovered, mutated child group) sticks in one generation.

To improve pair in same generation problem, I increased mutation rate a little from #12. It fixes little, as you can see from image, generation 2 has different 3th and 4th fitness at #12-3.
Also, since CX takes much time, so I reduced generation size to 20.

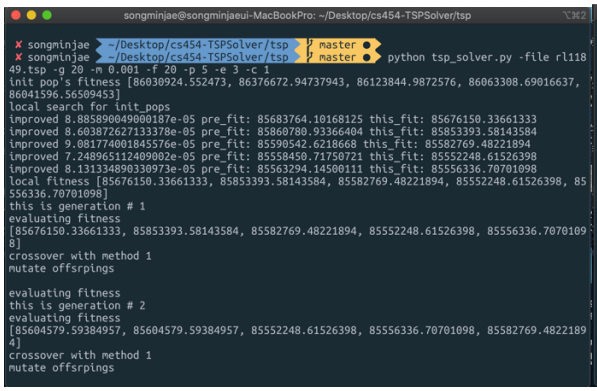
<Trial #12-1>
I tried normal crossover for comparison.
Population size = 5

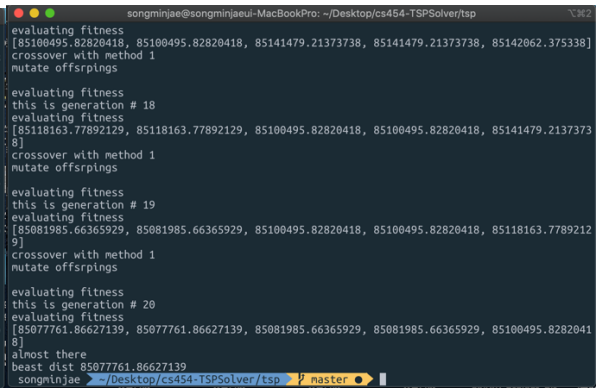
Fitness evaluation limit = 20(only for local search per one init individual)
Generation size = 20
Parent number = 2
Elite number= 3
Mutation rate = 0.001
Neighbor size = 50
Result = 83609480.49019258



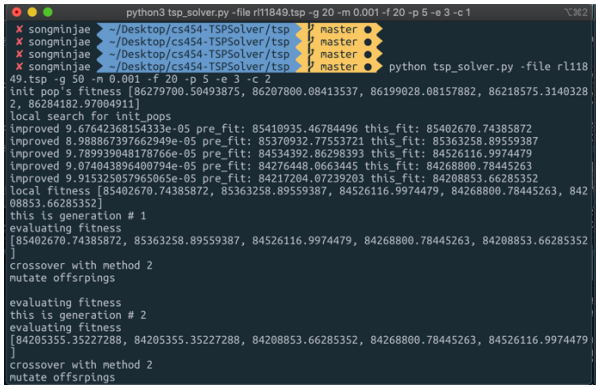


<Trial #12-2>
I also tried TMC
Population size = 5
Fitness evaluation limit = 20(only for local search per one init individual)
Generation size = 20
Parent number = 2
Elite number= 3
Mutation rate = 0.001
Neighbor size = 50
Result = 85077761.86627139
85552248.61526398





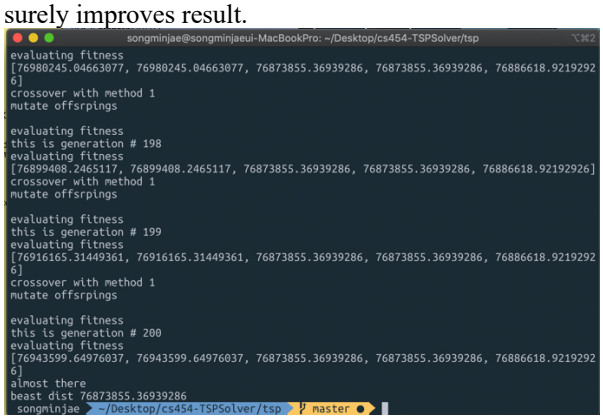
<Trial #12-3>
I also tried CX(cycle crossover)
Population size = 5
Fitness evaluation limit = 20(only for local search per one init individual)
Generation size = 20
Parent number = 2
Elite number= 3
Mutation rate = 0.001
Neighbor size = 50
Result = 83780285.66349736



Since, different from other crossover methods, CX does not depends on probabilistic. There is two possible crossover results in CX. Previously I put one result from two parent's crossover into child pool. But, for CX, I placed both results into child pool.

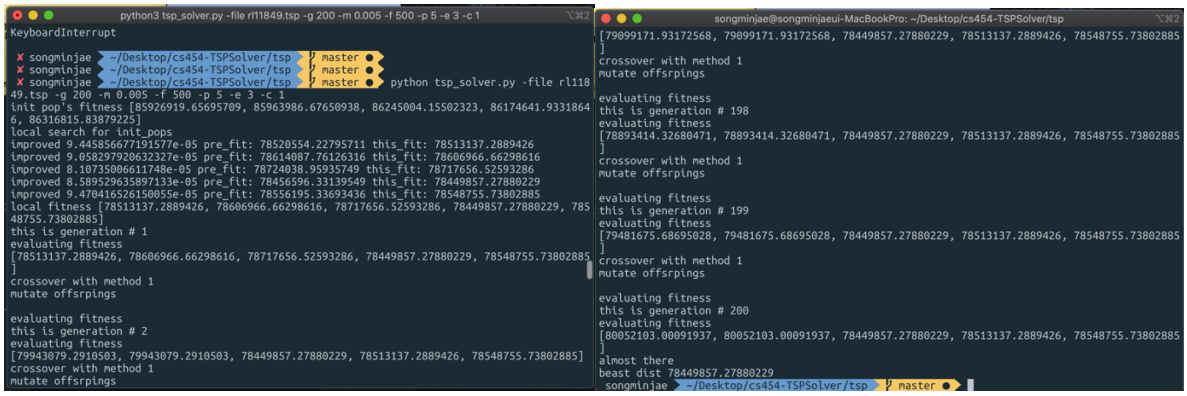
<Trial #13-1>
Population size = 5
Fitness evaluation limit = 500(only for local search per one init individual)
Generation size = 200
Parent number = 2
Elite number= 3
Mutation rate = 0.002
Neighbor size = 50
Result = 76873855.36939286

Used TMC. Comparing this with #9, we changed generation size from 50 to 200 and mutation rate slightly. It surely improves result.

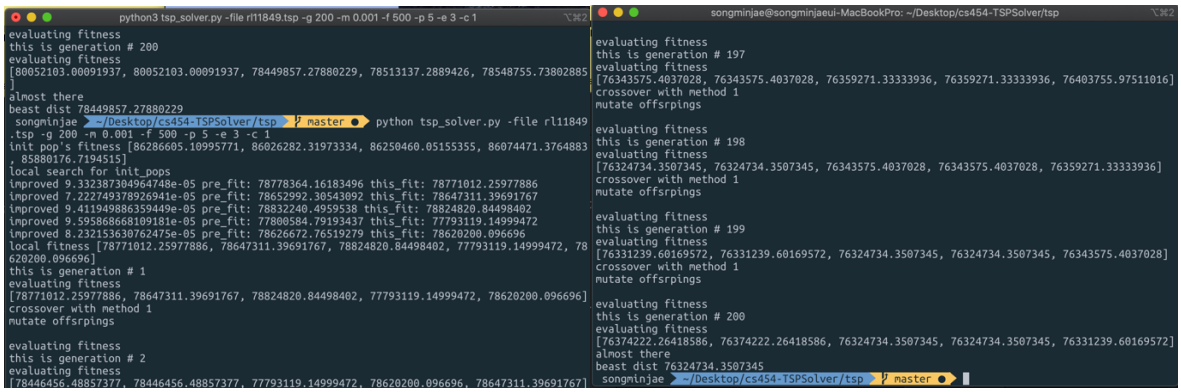


<Trial #13-2>
Population size = 5
Fitness evaluation limit = 500(only for local search per one init individual)
Generation size = 200
Parent number = 2
Elite number= 3
Mutation rate = 0.005
Neighbor size = 50
Result = 78449857.27880229

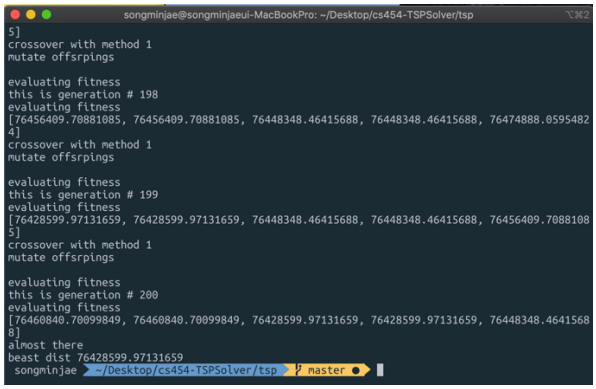
Mutation rate of 0.002 is better, for 0.005, child became less competitive and made whole generation deteriorate.



<Trial #13-3>
Population size = 5
Fitness evaluation limit = 500(only for local search per one init individual)
Generation size = 200
Parent number = 2
Elite number= 3
Mutation rate = 0.001
Neighbor size = 50
Result = 76324734.3507345

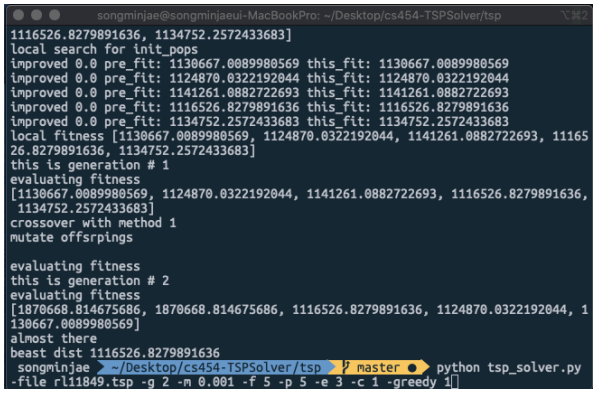


<Trial #13-4>
Population size = 5
Fitness evaluation limit = 500(only for local search per one init individual)
Generation size = 200
Parent number = 2
Elite number= 3
Mutation rate = 0.0005
Neighbor size = 50
Result = 76428599.97131659



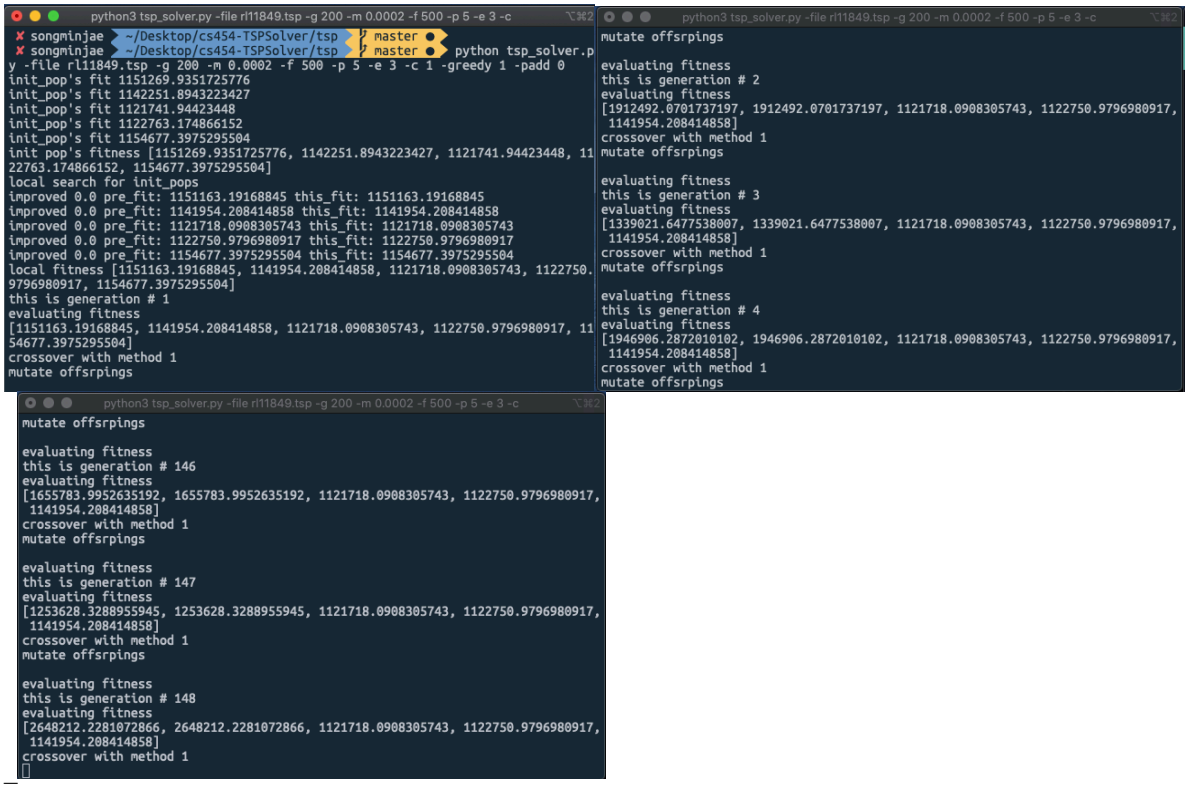
<Trial #14-1> I used greedy algorithm instead of padding.

Population size = 5
Fitness evaluation limit = 5(only for local search per one init individual)
Generation size = 2
Parent number = 2
Elite number= 3
Mutation rate = 0.001
Neighbor size = 50
Result = 1116526.8279891636



Greedy algorithm really improves much. But, mutation rate 0.001 is very bad for greedy cases because offspring(generation 2's 0,1 elements) have very low competitiveness.

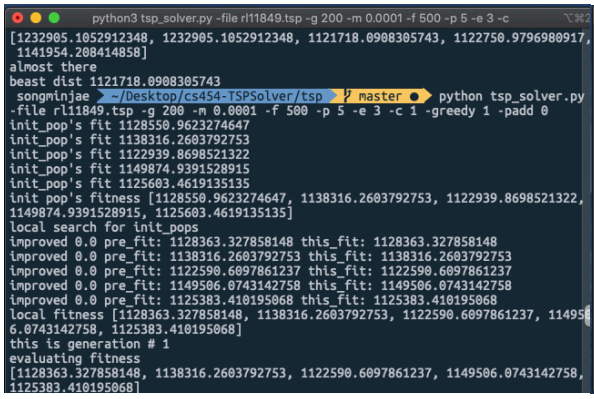
<Trial #14-2> with greedy, no padding
Population size = 5
Fitness evaluation limit = 500(only for local search per one init individual)
Generation size = 200
Parent number = 2
Elite number= 3
Mutation rate = 0.0002
Neighbor size = 50
Result = 1121718.0908305743



From first trial, changed mutation rate a lot. 0.0002 of mutation is about mutation of 2 pairs(11849 * 0.0002 = 2.3698).

Even reducing mutation rate does not change problem of less competitive child. As you can see above, sometimes mutation and crossover makes really high distance compared to peers in generation.

<Trial #14-2> with greedy, no padding. Greedy combined with padding is too much
Population size = 5
Fitness evaluation limit = 500(only for local search per one init individual)
Generation size = 200
Parent number = 2
Elite number= 3
Mutation rate = 0.0001
Neighbor size = 50
Result = 1122590.6097861237



Reducing mutation rate surely makes children a bit competitive.

Conclusion

Generational Selection

Comparing #4, #5 to #1,#2,#3 generational selection with elitism and gradual replacement makes little improvement.

Local Search

Local search improves distance much.

Appropriate threshold for improvement was about 0.0001. This caused about 30 -40 extra search when I placed fitness evaluation limit for 2000.

Local search is surely good optimization approach, but just putting it too much will not improve as expected. From my test, about 8000 number of local search makes improvement rate dull.

8000 local search for each individual, will improve init pop's fitness to about 25000000.

Crossover

From crossover part, we calculated improvement each since 12-1 has too good initial population in aspect of fitness. Improvement was calculated by

improvement = (generation #1 best – generation #20 best) / generation#1 best

Improvement(%)
#12-1 : 0.003352294585416
#12-2 : 0.00554616338755
#12-3 : 0.006269424578703

Results shows that CX is best at improvements. But, comparing #12-2, #12-3, CX takes too much time. So, when we consider "time consumed" for decision, TMC is not a bad choice.

Mutation Rate

As mentioned above, mutation rate 0.2 is too high, which makes children not competitive. But, also pushing rate into too low (0.0005), make same pair children in one generation problem. This will make population size meaningless.

From #13, we can find that high mutation rate like 0.005 makes uncompetitive child. Mutation rate less than 0.002 was effective for evolutionary computation.

Greedy Algorithm

Using greedy algorithm really makes results developed. But, for results about 1000000, mutation rate with 0.001(which was quite effective at 8000000-7000000 results) makes children really less competitive. From these we can see that higher mutation rate is less effective at more improved generation. The reason for this is that higher mutation rate makes more change in individual.

Greedy Algorithm makes good results and its effect is very high, so I think for more optimize, increasing population size will make results improved.

To short, from above results, I expect result to be best when I use [fitness limit(only for local search per one init individual) : 8000(if we do not use greedy… for greedy, 500 is enough), crossover : CX, mutation rate : 0.0001, gradual replacement, greedy approach]

Future Work

From my search about crossover methods, there are diverse methods improved from TMC and CX. From TMC, UTMX (uniform partially-mapped crossover), which uses probability of correspondence for each iteration and using it to slice genes. Also, CX2 which was introduced from paper may also improve fitness.

For testing, time cost was too much burden for me. Especially, CX crossover, widening neighbor size takes really much time for testing several times even though they improved fitness a lot. Generally, most of test were hard to execute multiple times. Approximately, for one individual's local search with fitness limit 2000 took about 33minute. This leads population initialization to take about 3hours when we have 5 population size.

From my friend's case, he had GPU server and ran his codes in parallel way. If I have some chance to ran in that kinds of condition, I want to run my code with parameters that I think will lead best result.

Also, bio-inspired algorithms like PSO, ACO may improve results.

I added greedy a bit late. So, it was hard to test greedy algorithm based hyper parameters. If I have more time, I want to test greedy added code much. Plus deviation between greedy and non-greedy is too much. Even many of my methods to improve results was useless to greedy algorithm. It was a little vain for me because I spent most of my time to improve non-greedy methods.

<Reference>

- Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator https://www.hindawi.com/journals/cin/2017/7430125/
- A Comparative Study of Adaptive Crossover Operators for Genetic Algorithms to Resolve the Traveling Salesman Problem https://arxiv.org/pdf/1203.3097.pd

details about parameters in my code.

Args

1. -file, --file_name
2. -p, --population_size, population size of one generation.
3. -f, --fitness_evalution, number of fitness evaluation for one individual's "local search"
4. -g, --generation_size, number of generation size in evolutionary loop
5. -parent, --parent_number, number of parent which crossover and makes offsprings
6. -e, --elite_number, number of elites in generation and elites remains after generational selection
7. -m, --mutation_rate, percent of mutated gene in mutation operation
8. -n, --neighbor_size, number of neighbor's size in local search
9. -c, --crossover_method, 0 for normal crossover 1for TMC and 2 for CX
10. -padd, --padding_number, number of padding when init population
11. -i, --improve_limit, limit for improvements
12. -greedy, --is_greedy

Number of distance evaluation is spreaded in my code.

distance evaluation executed

1. Local search => fitness_limit * pop_size * neighbor_size
2. init pop => padding * pop_size
3. evaluate fitness in evolutionary loop => pop_size * generation_limit
4. mutate_off => pop_size * generation_limit

+ Adding these 4 parts will result in total distance evaluation number.
Furthermore, there is improvement limit which extends distance evaluation. If improvement does not meet this value, they keep doing local search. If you want to turn off this functionality, just put enormous number to improve_limit(e.g. 100.1, 99.444…)

For simplicity, I made print for total call of “eval_fit” function.