School of Computing — Chansu Park

September 20, 2018 — 20173245

CS454
Fall 2018, Coursework #2

Here is the current record of my implementation:

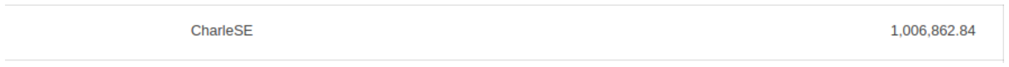
CharleSE 1,006,862.84

Figure 1: A sample result of the implementation.

1 Policies of this project

1.1 How to make and execute

This coursework has been built on `g++ 7.3.0` with `Ubuntu 18.04`. The language is `C++`.
There are three source codes: `City.cpp`, `tsp-solver.cpp` and `tsp-main.cpp` and the folder `tests` that contains some selected test cases.
You can find `Makefile` having three commands: `make all, make test, make clean`.
My implementation has 4 options which are mandatory:

- `-p` Decides how big the population is.
- `-f` Decides the number of the fitness evaluation. In my implementation, it equals to the number of generation.
- `-k` Decides how many genes will be kept without crossover. Should be smaller that the population.
- `-m` Decides how many mutation operator will be applied for non-best genes. In reality, the mutation will be reduced as the cumulated number of generation grows.

To parse the command line, I didn't use such headers like `unistd.h`, since it depends on the operating system. Instead I parsed manually.
You can find the sample execution command using `make test`. The name of the executable is `tsp-solver`. It accepts the file name (relative location) first and accept 4 following options, each with followed positive integer. Note that this command sends standard output to the file `output.txt`.

1.2 Parsing Input

Since the announcement said that we have to stick on the format that the sample input `fl11849.txt` has, I followed that one.
It has 6 lines of metadata: Name of the dataset, comment, type, dimension, edge weight

type, and node coordinate section.
Among them, I took a number at the line 4 (as `DIMENSION : number`) and used it as a size of the cities.
Following lines are the information of coordinates. Each line has one integer as an index of the city and two real numbers as an X/Y-coordinate.
So I ignored the first column and took the second and third number as coordinates. It doesn't matter whether or not it is represented by scientific notation.
I ignored "EOF" sign at the last line of the testcases since reading `<number>` lines is sufficient.

2 Implementation

I will only concentrate on `tsp-solver.cpp`. `City.cpp` has only helper functions about how to initialize a 2-dimensional point and computing distance between two cities; `tsp-main.cpp` just accepts the input and get a result from `tsp-solver.cpp`.
The main algorithm I used is Genetic Algorithm. I prepared such a size of pool, keep some best solutions with other optimization, and crossover the others using the best genes.

2.1 Initializing seed

I first constructed Minimum Spanning Tree (MST) of given complete graph. According to the Chapter 35.2 of [1], minimum spanning tree of graph with Euclidean metric induces a tour for Traveling Salesman Person (TSP) problem with 2-approximation. It is because Euclidean metric satisfies the triangle inequality. The method is intuitive:

1. Select a root vertex.
2. Compute an MSP using efficient algorithms and let the tree be T. (I used Prim's algorithm)
3. Perform a preorder tree walk on T starting from a root vertex. It generates an order H.
4. Return H, which is indeed an Hamiltonian cycle.

Thus, when I initialize genes, I first constructed an MST and copied them by `<population>` times. Since the MST is needed, I included some more data in my class `Genome`. Here are containers in `Genome`, the genes of my implementation:

`genome`	The container of the index of cities. TSP-tour of this instance will just follow the order in this container.
`parent`	Parent information of the minimum spanning tree.
`children`	Children information of the minimum spanning tree.
`key`	Helper array to construct the minimum spanning tree.
`minSpanTreeElmt`	The array for constructing minimum spanning tree.
`tspLength`	Contains the length of the given tour.

Now the initial pool has been constructed.

2.2 Improving the pool using GA

For each generation, my algorithm follow a simple loop:

- Compute the length of tour of each genes in the pool and sort it by the length.
- Among n genes, divide them into three classes: the best one $=: A$, other best $k-1$ genes $=: B$, the others $=: C$.
- Partially optimize genes in B.
- Crossover random genes in $A \cup B$ to replace C by the crossover result.
- Mutate and partially optimize genes in C.

Note that I also do some optimization not only for the crossover results but also for the other genes in B.

2.2.1 Partially-Matched Crossover

In [2], authors introduce a genetic operator called Partially-Matched Crossover (PMX). It accepts two genes as parents and outputs two childrens. The main idea is that, exchange

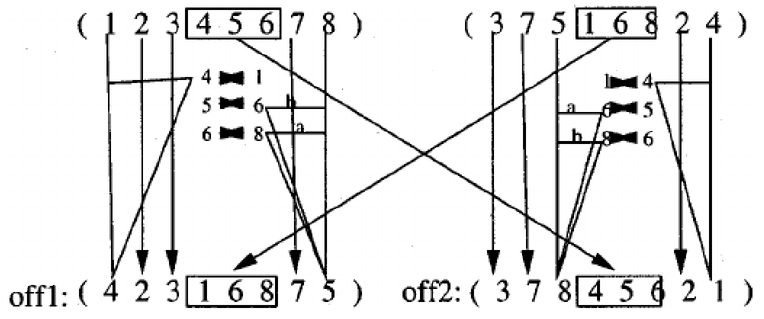

Figure 2: An illustration to explain how PMX works.

some fixed substring and put other elements into its original location. If the element already exists, then select the alternative by predefined mapping.
In the above illustration, 1, 6, 8 in the left side will be substituted by 4, 5, 6, respectively, and vice versa. Because of that, when the left side tries to put 8, it first maps 8 to 6, and since it still exists, it finally maps 6 to 5 and place that. Note that, if we put a sequence where every element uniquely exists in both parent, then the resulting sequences has every element uniquely. This operator works a little bit better than the order crossover operator introduced in the class.

2.2.2 Mutation using Double-Bridge move

To mutate the order, the easiest way is flipping a subsequence of the tour so that only two entrance and two exit changes. But it cannot introduce such a big mutation.
Double-Bridge Move ([3]) introduces a new movement: cross 4 substrings.
Suppose that there is a tour like $\{0, ..., A-1, A, ..., B-1, B, ..., C-1, C, ..., end\}$. Then this operator cut 4 edges $\{(A-1, A), (B-1, B), (C-1, C), (end, 0)\}$ and establish new 4 edges $\{(A-1, C), (end, B), (C-1, A), (B-1, 0)\}$.

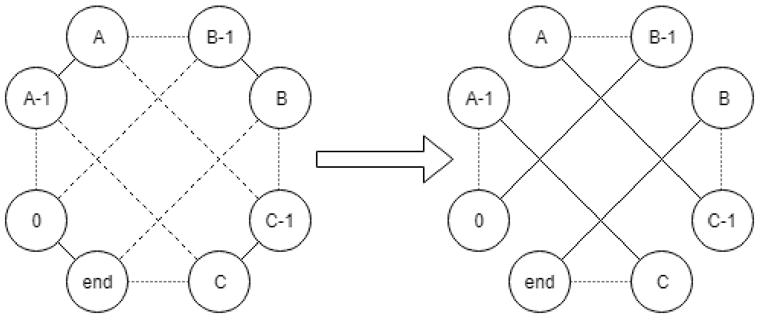

Figure 3: An illustration to explain how Double-Bridge Move works.

2.2.3 Partial optimization using 2-Opt

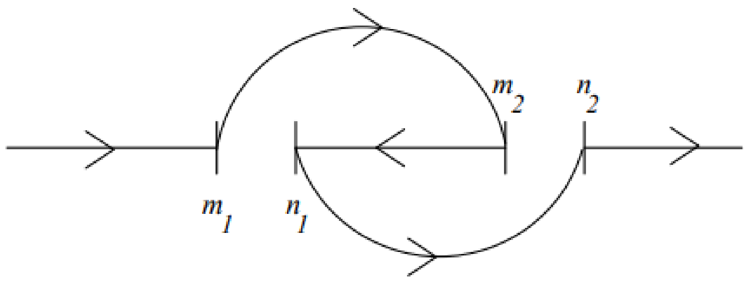

Figure 4: An illustration how 2-Opt performs its local optimization.

For crossovered genes and for non-best genes in the keeping list, I applied some partial optimization called 2-Opt. It is because the initial seeds are already too well-formed by MST. Performing only crossover and mutation will just increase their length in high probability so that they will just be deleted on the near future generation.
I performed an exhaustive search for every non-adjacent pairs of edges and applied 2-Opt operation up to the restricted mutation count.
To examine how many twoOpt operator will be applied to a single instance from MST, I measured the count using `rl5934.tsp` instance. It used this operator about 2000 times.

3 Experimental results

With the command `./tsp-solver ./tests/rl11849.tsp -p 40 -f 1500 -k 10 -m 25 > output.txt &`, I obtained the result up to 450 generations within 21 hours. It achieved 10 generations within first 5 minutes, but the speed has been decreased since 2-Opt takes more and more when the generation cumulates. (The order of searching pair of edges starts from the leftmost ones and goes to the rightmost ones.
Here are some intermediate results:

Generation 1 :1344568.40
Generation 5 :1334594.17
Generation 20 :1294286.18
Generation 50 :1235762.65
Generation 100 :1152052.33
Generation 200 :1006862.84

And I also noticed that the length didn't decrease from Generation 193. It seems that 2-Opt optimization for solutions reached a limit so that no optimization will occur for the best k keeping parents, and the partial optimization for children may not exceed the parents.
For the smaller instances like `pr76.tsp`, some noticible breakthroughs has occurred at once per 100-200 generations. If I started a little bit earlier, then I might be able to performing more intense optimization for children will help them to break parent pool and test them. Or maybe some breakthrough will be present after some more generation. I inserted `debug-rl11849.txt` for the instance for the leaderboard and `debug-pr76.txt` for the `pr76.tsp` instance in the folder `./tsp`.

4 To debug and find intermediate results

In `tsp-main.cpp` and `tsp-solver.cpp`, there is a define statement: `#define DEBUG ....` If you set this as 1, it prints every 5th intermediate tours and length for every generation, so you can observe them for large dataset. It also picks its metadata.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms. MIT press.

[2] Goldberg, D. E., & Lingle, R. (1985, July). Alleles, loci, and the traveling salesman problem. In Proceedings of an international conference on genetic algorithms and their applications (Vol. 154, pp. 154-159). Lawrence Erlbaum, Hillsdale, NJ.

[3] Martin, O., Otto, S. W., & Felten, E. W. (1991). Large-step Markov chains for the traveling salesman problem.