Contents

1 Intro

This is report for a hw01 for EOA course on CTU FEE. The goal has been to try to solve traveling salesperson problem by means of evolutionary algorithms.

The traveling salesperson problem tries to find the shortest possible route when traveling through multiple cities, visiting every city exactly once, returning to the origin city.

The report covers the implemented algorithms and the results on 10 TSP instances from TSPLIB. All of those chosen instances are using euclidian metric and are 2D.

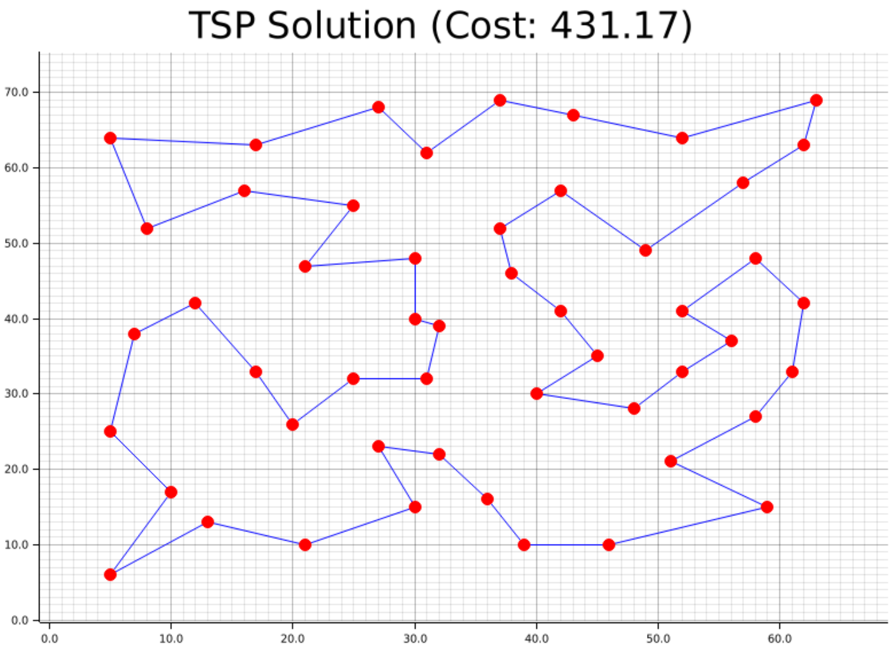

Figure 1: eil51 instance candidate solution found by EA with nearest neighbor heuristic.

2 Representations

Two representation have been chosen for the TSP problem,

1. Node permutation
2. Binary string half-matrix denoting if city i comes before city j

2.1 Node permutation

Implemented as a vector of indices of the cities. Couple of perturbations and crossovers have been implemented:

- Perturbations
  - Move single random city to random position
  - Swap two randomly selected cities
  - Reverse subsequence of cities (random starting point and random length)
- Crossovers
  - Cycle crossover
  - Partially mapped crossover
  - Edge recombination crossover

Also apart from a random initializer, two additional initializers are implemented, one based on minimal spanning tree and second based on nearest neighbors. Detailed descriptions of the algorithms follow in next sections.

2.1.1 Crossovers

All used crossovers take two parents and it's possible to create two offsprings out of the two parents by switching the parent positions. (switching first parent and second parent)

2.1.1.1 Cycle crossover The cycle crossover creates a cycle, takes parts of the first parent from the cycle and fills the rest from the second parent.

The cycle is created as follows:

1. Start with index 0, call it current index
2. Save current index
3. Look at current index in first parent, let's call it current element
4. Find the same element in second parent and update current index to this new index
5. Repeat 2 - 4 until reaching index 0 again

Then the offspring is created as follows:

1. Clone the second parent
2. Iterate the saved cycle indices, take element at index from first parent and copy it to offspring at the same index

2.1.1.2 Partially mapped crossover The partially mapped/matched crossover randomly selects two points. At the end it should ensure that the offspring has first parent's elements in between the two cross points.

The way to ensure that, while still ensuring the result is a valid permutation, is to always swap the elements.

Offspring is created as follows:

1. Clone second parent
2. Then, for every index i between the cross points:
3. Take the element on index i from first parent
4. Find the city in the offspring, let's call its index j
5. Swap i with j

2.1.1.3 Edge recombination crossover Edge recombination is the most complicated from the three crossover operators.

First, an adjacency list is created for both parents. Let's take an example of a permutation: 012345. The element 0 has 5 and 1 as adjacement nodes. 1 has 0 and 2 and so on. The two adjacency lists are then joined together, while omitting repetitions. That means that a single element will have from 2 to 4 elements in its adjacency list.

To make an offspring:

1. First element is taken out of the first parent, this is the current element (city).
2. The current element is appended to offspring
3. The current element is removed from adjacency lists of all other elements.
4. Then, all the cities in adjacency list for current element are taken and one with minimum neighbors in its adjacency list is taken. If there are mutliple such cities, random one is chosen.
5. The found city becomes current element.
6. Steps 2 to 5 are repeated until all elements are taken.

In the code, the algorithm has been implemented slightly differently. Instead of removing any elements, the elements are just marked as removed. Also, instead of constructing a list of adjacency nodes progressively, a vector of 4 elements is made beforehand. Each element may be unset (an Option). Then, the adjacencies from first parent are put to positions 0, 1 and adjacencies from second parent to positions 2 and 3. If there would be a repetition, it's left unset. This then allows for static allocation only. This is thanks to a library called nalgebra is used and it's possible to tell it the dimensions of adjacency matrix beforehand. However for the runs of the algorithms, `Dyn` dimension has been used out of convenience, so dynamic allocation is performed.

2.2 Binary string

Classical perturbations and crossovers have been implemented for the binary string representation, specifically:

- Perturbations
  - Flip each bit with probability p
  - Flip single random bit
  - Flip of N bits (N is chosen beforehand, not randomly)
  - Flip of whole binary string
- Crossover
  - N point crossover

As for initialization, random initializer has been implemented.

The fitness function is implemented by form of a wrapper that converts the BinaryString into NodePermutation and then the same fitness function is used as for node permutation representation.

2.2.1 N-point crossover

The N point crossover works on two parents and is capable of producing two offsprings. The crossover first chooses N cross points randomly.

Then, the cross points are ordered in an ascending order and first all bits from first parent are chosen until cross point is encountered, then all bits are taken from second parent until another cross point is reached. And this repeats until the end of the string is reached.

Also, one-point and two-point crossovers have been implemented separately for more effective implementation than the generic N-point.

3 Prepared algorithms

Three generic algorithms have been used for solving the TSP. Then various operators have been tried with each of them.

3.1 Evolution algorithm

The algorithm keeps a population of N each epoch and generates offsprings out of the current population. Specifically it uses the following steps:

1. Obtain initial population (ie. random or specialized heuristic).
2. Select parents to make offsprings from (selection).
3. Pair the selected parents (pairing).
4. Somehow join the parent pairs to make an offspring (crossover).
5. Replace the current population with a new one, combining the current population and offsprings. (replacement)

6. Repeat steps 2 to 5 until terminating condition is reached (number of iterations)

During its run the algorithm saves the best candidates encountered. It saves them even during one epoch for better granularity. This also ensures that if best candidate does not make it into the next population, it is still captured by the algorithm and can be returned as best candidate at the end. The implementation is in `eoa_lib/src/evolution.rs`.

3.2 Local search

The local search implementation is implementation of the first improving strategy, where the current best solution is being perturbated, until a better solution is found. And then, the process starts over from this better solution.

The local search implementation supports evolutionary strategies, ie. changing the perturbation paramters during the run, but none were used for TSP.

The implementation is available in `eoa_lib/src/local_search/mod.rs`.

3.3 Random search

Random search initializes new random elements in the search space each iteration and saves new elements if they are better than best found.

The implementation is available in `eoa_lib/src/random_search.rs`

4 Heuristics

Instead of starting with random solutions, two heuristics have been tried to make the initial populations for the evolutionary algorithms.

4.1 Nearest neighbors

The heuristic starts at a given node and finds it's nearest neighbor. Then adds that neighbor to the permutation and moves to it. Then it repeats search for the nearest neighbor, making sure to not select nodes twice. The whole chromosome is built like this.

Moreover the possibility to select second neighbor instead of first has been incorporated in the heuristic as well. Specifically, it's possible to choose the probability to choose second neighbor instead of first one.

This makes it possible to generate a lot of initial solutions, specifically it's possible to generate nearest neighbors starting from each city and then it's possible to tweak the probability of choosing the second neighbor.

To initialize the whole population: 1. Generate chromosome starting from each city, choosing the first neighbor. 2. Generate chromosome starting from each city, but always choosing the second neighbor. 3. The rest of the population is initialized from randomly selected cities with randomly selected probability of choosing second neighbor.

4.2 Minimal spanning tree

For using a minimal spanning tree the algorithm of Christofides and Serdyukov has been chosen as an inspiration. But instead of trying to find an Eulerian tour on the minimal spanning tree, a slightly different randomized approach has been taken. Specifically, random node is chosen as the starting point and then edges are chosen out of the minimal spanning tree to go through. If there are more edges to travel through, random one is chosen. If there are no edges left, nearest neighbor is chosen as next node.

To initialize the whole population, the same algorithm is ran multiple times. Since it is random, its results should be different, at least slightly.

5 Results

To compare all the algoritms on various instances, always at least 10 runs of the algorithm have been made on the given instance. All the graphs of probabilities of reaching target for given function evaluation were then constructed from averaging between the runs. The fitness charts sometimes show less runs to not be too overwhelming. All evolution algorithms ran on 5000 iterations (epochs) and the local search and random search were adjusted to run on the same number of fitness function evaluations. The population size for the evaluation algorithm 500 has been chosen. The number of parents (and thus offpsrings) is 250.

The instances chosen from TSPLIB, all 2D Euclidean:

- eil51
- eil76
- eil101
- kroA100
- ch150
- kroA150
- kroA200
- a280
- u574
- u724

5.1 Comparing perturbations on LS

Four perturbations have been evaluated: - Moving single city - Swapping of two cities - Reversing subsequence - Combination of the above (specifically single random perturbation has been chosen to run)

5.2 Comparing algorithms

To compare the algorithms, first it has been ensured the algorithms were tweaked to produce the best results (best that the author has been capable of). Then, they were ran on 10 instances of TSP and averaged in the following chart:

5.3 Comparing crossovers

During evaluation of the various crossovers, it has become apparent that with the currently chosen parameters of the algorithms, the dominant parts that are responsible for the good convergence are: selection, perturbation and replacement, but not the crossover itself. In other words, just replacing the crossovers produced similar results. For this reason, the mutations probabilities have been lowered significantly for this comparison. This then allows for comparison between the crossovers themselves.

From the runs, the edge recombination has produced the best results, with partially mapped crossover being the second and cycle crossover being the worst.

5.4 Comparing representations

5.5 Comparing heuristics

Both of the heuristics are capable of making initial solutions much better than random ones. Here is a fitness graph with only the two heuristics.

And here is a probability (TODO)

From the results it can be seen the nearest neighbor heuristic performs better. But this could be caused by the fact that minimal spanning tree has been used along with a randomized approach. If more deterministic approach of finding the best Eulerian path has been chosen, it's possible it would perform better.

6 Things done above minimal requirements

- 1 point - Compared 2 representations (permutation and binary string with precedences)
- 1 point - Compared 4 LS perturbations (swap, reverse subsequence, move city, combination)
- 1 point - Compared 3 crossover operators (edge recombination, partially mapped, cycle)
- 1 point - Compared all algorithms on 10 instances
- 1 point - Initialized with two constructive heuristics (nearest neighbors, solutions from minimal spanning tree)

7 Code structure

Rust has been chosen as the language. There are three subdirectories, `eoa_lib`, `tsp_hw01` and `tsp_plotter`.

`eoa_lib` is the library with the generic operators defined, with random search, local search and evolution algorithm functions. It also contains the most common representations, perturbations and crossovers for them.

`tsp_hw01` contains the TSP implementation and runs all of the algorithms. It then produces csv results with best candidate evaluations for given fitness function evaluation count. This is then utilized by `tsp_plotter`. The node permutation tsp representation itself is implemented in `tsp.rs`. The configurations of all algorithms used are located in `main.rs`. All the instances used are located in `tsp_hw01/instances` and the solutions are put to `tsp_hw01/solutions`

`tsp_plotter` contains hard-coded presets for charts to create for the report.

8 Usage of LLM

While I was working on this homework, I have used LLM for writing certain parts of the code, specifically I have used it very extensively for the tasks that I do not like to do much myself: - Loading data, - Plotting graphs, - Refactoring of a triviality at a lot of places at once (ie. at first I chose to make perturbation copy the chromosome, but I realized this is very ineffective for EAs afterwards and changed it to change in-place instead), - Writing some of the tests

As I am not proficient with Rust, sometimes I asked LLM to help with the syntax as well, to find a library that will help solve a task or what functions are available for a specific task.

I have used LLM only minimally for implementing the algorithms or for deciding on how to make the implementation, mainly sometimes in the form of checking if the algorithm looks correct. (and it did find a few issues that I then sometimes let it fix and sometimes fix myself). This is because I believe that I can learn the most by writing the implementations myself.

I use Claude from within claude-code, a CLI tool that is capable of reading files, executing commands and giving the model live feedback, like outputs of ran commands. This means the LLM is able to iterate by itself, without my intervention, for example when fixing errors. On the other hand it can use a lot of tokens quickly unless I make sure to run compact often.