Parallel genetic algorithms library

The library gives an interface to solve Traveling Salesman Problem via genetic algorithms

1st Dmytro Shumskyi
Applied Science Faculty
Ukrainian Catholic University
Lviv, Ukraine
shumskyi.pn@ucu.edu.ua

2nd Marta Samoilenko
Applied Science Faculty
Ukrainian Catholic University
Lviv, Ukraine
samoilenko.pn@ucu.edu.ua

3rd Victor-Mykola Muryn
Applied Science Faculty
Ukrainian Catholic University
Lviv, Ukraine
muryn.pn@ucu.edu.ua

4th Yurii Sahaidak
Applied Science Faculty
Ukrainian Catholic University
Lviv, Ukraine
sahaidak.pn@ucu.edu.ua

5th Roman Milischuk - Mentor
FoxFour CTO
King's College
London, United Kingdom
roman@foxfour.ai

***Abstract*—There are plenty of NP-complete algorithms, which require a lot of time to be solved. The genetic algorithms allow us to reach great results with significantly lower complexity. The project's main topic is the development of different techniques for genetic algorithms and comparing them with each other. The Traveling Salesman Problem (later TSP) is an NP-hard problem. The most popular exact algorithm - Held-Karp algorithm has complexity $O(n^2 2^n)$.**

***Index Terms*—genetic algorithms, traveling salesman problem, combinatorial optimization**

I. Introduction

A. Genetic Algorithms overview

The genetic algorithms use the natural selection principle to optimize selection tasks. The principle relies on three main processes: crossover, mutation, and selection. The genetic algorithms manipulate populations of possible solutions. Via modifications of existing individuals and selecting the best ones, the algorithm obtains pretty accurate results, conducting significantly fewer operations than exact algorithms.

B. Traveling Salesman Problem

The TSP is a problem related to searching for cycles through each graph vertex exactly once with the smallest weight. As long as the problem belongs to the NP-hard class, the solution's complexity increases faster than any polynomial but still slower than $O(n!)$. The problem appears in many spheres: logistics, astronomy, DNA sequencing, etc. Thus, a need for a fast but not exact solution arises.

C. Held-Karp algorithm

The Held-Karp algorithm is one of the most popular TSP solutions. Its complexity is $O(n^2 2^n)$, which is much better than the complexity of brute force $O(n!)$. Still, the Held-Karp algorithm uses more memory $O(n2^n)$ to solve the problem. It stores only data about the shortest paths from start to point through a set of points. This leads to skipping unnecessary operations but still getting exact results. Thus, the algorithm will be used as a genetic algorithm's results reference value.

II. Genetic Algorithms in detail

A. Population

The population is a set of individuals representing the problem's solution. In the case of TSP, it is a set of permutations of vertexes that describe the order of vertexes in the cycle. The first population is randomly generated, while each other is based on the best individuals of the previous one.

B. Crossover

The crossover is the process of mutation in a genetic algorithm. Two randomly chosen individuals exchange parts of their genes, in the exact case, parts of vertex permutations. The crossover can be conducted in different ways. We will consider only crossovers that work with permutations of the same sets and are applicable to TSL problem. For instance, the **Order crossover** takes part from the first individual and sets it to the permutation of the other one. Then it fills empty positions with elements from the set in the same order as in the second individual. The **Partially mapped crossover** randomly takes elements from the first individual. Then it looks at elements of the second individual, position of which are now occupied, and sets them to positions of elements that occupied the position. All the other elements are copied from the second individual's permutation. The library has 2 built-in crossover algorithms: Order crossover (default; in the library, it is called a simple crossover) and Uniform crossover.

1) Default crossover (Order crossover): The default crossover algorithm is the Order crossover. It selects a random subset of the first parent and copies it to the child. Then, it fills the child with the elements from the second parent in the order they appear in the second parent. The remaining elements are copied from the second parent in the order they appear in the first parent.

2) Uniform crossover: The Uniform crossover selects a random subset of the first parent and copies it to the child. Then, it fills the child with the elements from the second parent

in the order they appear in the second parent. The remaining elements are copied from the second parent in the order they appear in the first parent.

C. Mutation

The mutation is a process of random change in some permutations. While crossover is a binary function, the mutation is unary. As a crossover, a mutation can be made in different ways. For instance, **Rotation to the right** selects a subset of the permutation and rotates all elements in it by some value. **Inversion** as a Rotation selects a subset, but instead of shifting elements, it inverses their order.

The library has 3 built-in mutation algorithms: Swap mutation (default; in the library, it is called a simple mutation), Inversion mutation and Rotation mutation.

1) Default mutation (Swap mutation): The default mutation algorithm is the Swap mutation. It selects two random elements of the permutation and swaps them.

2) Inversion mutation: The Inversion mutation selects a random subset of the permutation and inverts the order of elements in it.

3) Rotation mutation: The Rotation mutation selects a random subset of the permutation and rotates all elements in it by some value.

D. Selection

The selection process determines the best individuals for the next generation. The individuals should not be the best ones to avoid local minimums. Thus, the selection takes the best of some individuals several times.

The library now has 4 built-in selection algorithms: Tournament selection (default; in the library, it is called a simple selection), Rank selection, Boltzmann selection, and Proportional selection.

1) Tournament selection: From the population, it selects *k* individuals, then finds one the best of them and returns him.

2) Rank selection: This algorithm sorts the population in descending order (firstly, go better individuals). Then, it calculates a distribution, where the first members have higher priority than the last.

3) Boltzmann selection: This selection mechanism applies principles from simulated annealing to control selection pressure thermodynamically. The temperature is preset to control the selection rate and starts at a high level, resulting in lower selection pressure. Over time, the selection pressure increases, and individuals with better fitness are favoured in the selection process.

4) Proportional selection: In this algorithm, each Individual has a probability of being selected:

$$P_i = \frac{\text{fitness[i]}}{\sum_{k=1}^{n} \text{fitness[k]}} \tag{1}$$

The algorithm generates a distribution and randomly selects an Individual from that distribution.

III. Library interface

The library [1] includes three fundamental classes that define the API.

A. Individual

Individual is an abstract class that has 2 required attributes (solution and fitness) and 1 method (spaceship operator). The solution is a vector of type size_t. Each number could represent something at the discretion of the user. Fitness is a measure of how good the Individual is.

B. Population

Population is an abstract class that other classes should inherit. This class comprises four attributes (population and selection, crossover, and mutation types) and methods that implement default algorithms. To select a different algorithm, use the *setCrossover* (*setSelection*, *setMutation*) function and pass the required algorithm. Available algorithms are stored in corresponding enums (*selections*, *crossovers*, *mutations*). If a particular algorithm is unavailable, the user should override the corresponding function (further details to follow).

Additionally, this class features three more methods: *evaluate*, *isFirstBetterThanSecond*, and *getBest*. *isFirstBetterThanSecond* a function that says which of two Individuals is better (with the higher fitness by default, but the user can override this function). *getBest* returns the best Individual from the population according to the previous function. The user can override them both if he needs to. *evaluate* should return the fitness of an Individual. The user should override this function for their purposes.

C. Solver

Solver — a class that makes logic to solve some genetic problem. The constructor takes a SetUp struct as the only argument. It has four fields: generation number, mutation rate, crossover rate, and sorted. Those rates describe how many mutations and crossovers will be performed. The rates must be [0, 1], and their sum must be ≤ 1. The generation number is the number of iterations that will be done. Sorted (default false) means sorting the population after each generation. For instance, it could be useful for rank selection (if the user selects some default selection/crossover/mutation that requires sorting, the sorted attribute is set to true automatically; there is no need to change it manually).

To solve a problem, use the solve method. This method makes *generationsNum* iterations. On each iteration, we make crossoverNum of crossovers, mutationNum of mutations, and update the current population.

D. Example

To solve the problem, the user needs to create a new population class, which extends from the Population. In the constructor, he should initialize the population: make Individuals, and evaluate their fitness. Also, the user needs to override the evaluate function for his purposes.

If the user finds it useful, he can override mutation, crossover, and/or selection to use his algorithm or one of the existing ones. To do that, the user calls a setSelection (or other corresponding function) method of the extended population.

To run solving, the user calls the solve method of Solve instance and gives an extended population as an argument. The result will be the best Individual after evolution.

IV. Parallelization

To run in parallel, our library uses Thread Pool. Since all crossovers, mutations, and selections in the solve method could be done independently, the library performs them in parallel. By default, it uses all available hardware threads, so, in general, the parallel version will run 6-8 times (depending on the number of threads on the user's computer) faster than in 1 thread.

V. Testing

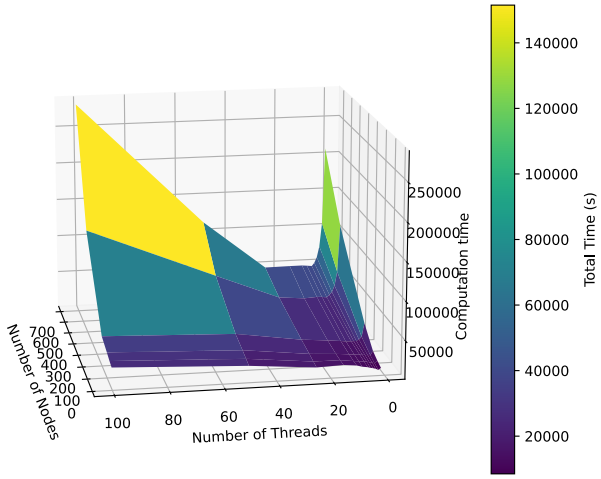
To test this library, we chose the Travelling Salesman Problem [2]. The exact solution to this problem can be found in $O(2^n n^2)$ using the Held-Karp algorithm, which means every additional point on the graph will double the time up. There is no sense in waiting for a solution of more than 25-30 points. Also, this implementation of the Held-Karp algorithm is parallel and uses Thread Safe Queue. But actually, this allows us to add only 1 extra point to the graph.

At the same time, this library works with graphs of sizes 100 and even 1000. We could control the execution time and precision using SetUp parameters and generation numbers.

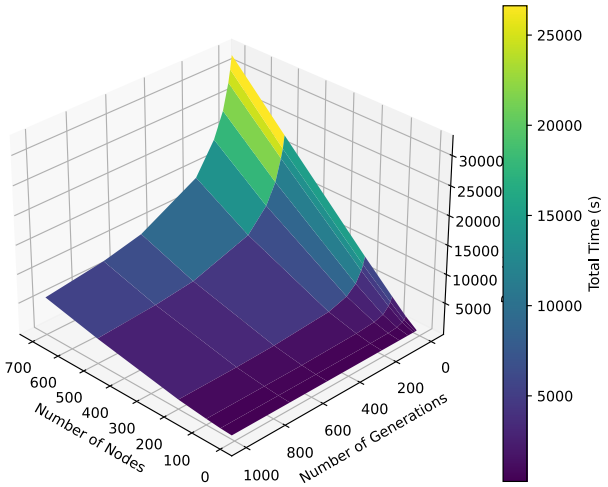
Also, you can find an example of the Knapsack Problem on GitHub.

VI. Results

A. Time complexity



This plot shows that the time complexity is linear to the number of nodes if you use the right number of threads.



Also, this plot shows that our algorithm has a linear dependency on the number of generations.

This leads us to conclude that time complexity $O(n * d)$, where n is the number of nodes and d is the number of generations.

B. Parallelization level

We will use Amdahl's law to identify how many percent of the programs are parallelized.

$$T(n) = T(1-p) + \frac{Tp}{n} \tag{2}$$

- T - computation time
- p - parallel part
- n - number of threads

The equation is a Polynomial of -1 power. Thus the following system can be created.

$$\begin{pmatrix} 1 & \frac{1}{n_1} \\ 1 & \frac{1}{n_2} \\ 1 & \frac{1}{n_3} \\ \dots & \dots \\ 1 & \frac{1}{n_{10}} \end{pmatrix} \begin{pmatrix} T(1-p) \\ Tp \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ \dots \\ t_{10} \end{pmatrix}$$

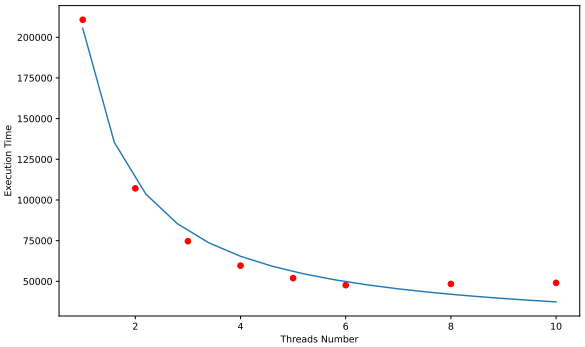
A normal equation can be applied to find the best least square solution of the polynomial. The solution is found in bound of 10 threads because the machine which run the program has only 6 cores with 12 logical cores.

$$A^T A x = A^T b \tag{3}$$

Solving those linear equations gives us the following results.

$$\begin{pmatrix} T(1-p) \\ Tp \end{pmatrix} \begin{pmatrix} 18692.2 \\ 186841.3 \end{pmatrix}$$

So $T = T(1-p) + Tp = 205533.5$, and $p = \frac{Tp}{T} \approx 0.9$. So, the level of parallelization is 90%.



Also, here is a plot of this approximation. The red dots are the results of running, and the blue line is a predicted time, given that the parallelization level is 90%.

VII. Conclusion

The library provides an interface to solve the Genetic Algorithm and Knapsack problems. It is easy to use and has a lot of built-in algorithms. The library is parallelized and uses all available hardware threads. The library has been tested on the Travelling Salesman Problem and has shown good results. Time complexity is linear in terms of the number of nodes and generations. The level of parallelization is 90%.

VIII. Acknowledgment

We would like to thank Roman Milischuk for his help and guidance during the project.

References

[1] DmShums. (2024). GitHub - DmShums/ga_lib. GitHub. https://github.com/DmShums/ga_lib

[2] Shendy, R. (2024, January 28). Traveling Salesman Problem (TSP) using Genetic Algorithm (Python). Medium. https://medium.com/aimonks/traveling-salesman-problem-tsp-using-genetic-algorithm-fea640713758

[3] Mitchell, M. (1996). An Introduction to Genetic Algorithms. Cambridge: MIT Press.