CS454 Assignment 02 Report

redacc (2019redacc)

April 16, 2020

1 Overview

This report outlines the following for KAIST CS454 Assignment 2 developed by redacc:

1. Approach to the TSP Problem
2. Build Instructions
3. Execute Instructions
4. Self-Test Methodology and Results
5. Structure of Program with brief explanations

Each of the items will be answered in their own sections.

2 Approach to the TSP Problem

The following subsections detail the process in how I decide to implement the solver for the TSP problem.

2.1 Choice of Implementation Programming Language

As the TSP problem is NP-hard problem that is considered a benchmark for optimisation, I decided to use C++ as the language of the implementation as it is considered one of the faster languages and every reduction of speed would be important for larger TSP problems.

2.2 Choice of Optimisation Algorithm

As the assignment restricted choices to be that of stochastic optimisation, I researched various meta-heuristic algorithms and very quickly came to the conclusion that a Genetic Algorithm will be the best algorithm for the job as various papers mention the Genetic Algorithm to be one of the faster TSP solvers when combined with a good Local Search optimisation alogorithm. It was also seen that selection wheel mating were one of the more optimal choices for selection.

2.3 Choice of Local Search Optimisation Algorithm

There is widespread consensus that the LKH (Lin-Kernighan Heuristic) is the best local search optimisation algorithm that exists for the TSP problem. However, after further review, I decided to utilise the 2-opt local search algorithm due to the potential for misimplementation of the LKH algorithm due to it's complexity and lack of testing with the C++ language as most implementation snippets are shown to be in Java. I also believe that utilising 2-opt would be faster when attempting to optimise each generation to allow for more genetic mating and subsequently allow for lucky reductions in travel distance.

3 Assignment Build Instructions

The project can be built by calling the `make` utility present in most unix terminals. Issuing a `make` command on terminal will automatically compile and link object code to generate an executable. I list the following `make` targets for specific build requirements.

1. `make` or `make all`
   This target will generate all object files for `tsp` and link them into the tsp executable. By default, the `make` command is the equivalent to `make all`. This target will usually be the one used for most build requirements.

2. `make clean`
   This target will remove all generated object files, log files and executables related to the tsp executable.

4 Assignment Execute Instructions

The following command and brief explanations on the parameters provide the executables with the correct functionality as per the project specification and some extra functionality.

1. `./tsp {filename} [-pflerst parameter]`

   The above executes the TSP solver with initialising information that are passed in as parameters. They are explained in further detail below. All possible parameters must be included for the program to function correctly.

2. `-p {population_size}`

   The above takes in a population_size integer that will be the amount of chromosomes that each generation will have. I suggest this number be atleast 30 at the minimum but it can be made smaller with bigger problem sets.

3. `-f {fitness_evaluations}`

   The above takes in a fitness_evaluations unsigned long long int that will be the total amount of fitness evaluations that the program will perform. If the total is reached during optimisation of a generation, it will stop optimisation right away, write the best solution found to the `solution.csv` file and output the best distance found. I suggest this number be 20 generations multiplied by the {local_threshold} and {population_size}.

4. `-l {local_threshold}`

   The above takes in a local_threshold integer that will be the amount of fitness evaluations during the local optimisation of each chromosome in the generation. I suggest this number to be 25000 to allow for an adequate amount of local optimisation.

5. `-e {elite_rate}`

   The above takes in a elite_rate float that will be the percentage (out of 100%) of the population size that will be automatically in the new generation. I suggest this number to be 20 to allow for better results to be shielded from the inherent randomness of Genetic Algorithms.

6. `-r {mutation_rate}`

   The above takes in a mutation_rate float that will be the percentage (out of 100%) that a select chromosome will be subject to a mutation. The mutation will be further detailed in the function responsible for mutation. I suggest this number to be 20 to ensure some randomness for the Genetic Algorithms so a local minimum is not reached if possible.

7. `-s {mutation_size}`

   The above takes in a mutation_size float that will be the percentage (out of 100%) of the chromosome size that will be subject to a mutation when chosen for mutation. The mutation will be further detailed in the function responsible for mutation. I suggest this number to be 20 to ensure some randomness for the Genetic Algorithms so a local minimum is not reached if possible.

8. `-t {warn_threshold}`

   The above takes in a warn_threshold integer that will be the amount of times that a non-improvement of the best result during the local optimisation per generation will be ignored per solve instance. I suggest this number to be 10 to allow for a potential local minimum that may be able to be skipped by the inherent randomness of the Genetic Algorithm.

5 Self-Test Methodology and Results

In order to ensure that the program meets and performs to the definitions of the specification, some test cases such as the `att48.tsp` instance was run using the above commands with the suggested parameters. The results approached the best known optimal solutions but were unable to completely match them. Sometimes it missed the global optimum by 300% outlining the luck involved to reach the global optimum.

6 Structure of Programs

6.1 Source Files

1. `tsp.cpp`
   This file contains the code that will read in the data from a chosen file (by parameter), process the data and initialise the genetic algorithm routine.

2. `matrix.cpp`
   This file contains the code that will process the data from the chosen file into a vector of `city_node` objects essential for the running of the genetic algorithm routine.

3. `ga_frame.cpp`
   This file contains the code that will establish a genetic algorithm framework and execute an genetic algorithm instance that will attempt to solve the given TSP Problem via finding the best path to travel all the nodes in the vector of objects provided by the matrix program.

4. `matrix.hpp`
   This header file contains the function prototypes and class declarations that are useful and common to both `tsp` and `ga_frame` source code files.

5. `ga_frame.hpp`
   This header file contains the function prototypes and class declarations that are used in the ga_frame source code. There is a header guard in place due to some overlap with `matrix.hpp`, namely the `city_node` class.

6. `Makefile`
   This file allows the `make` utility present in most unix-based terminals to compile and link the executable for `tsp` automatically.

6.2 Source Functions

This section will briefly outline some important functions present in the code. More detailed information on the functions are available as inline comments in the source files.

6.2.1 Debug

1. `void print_population(std::vector <int> population)`
   This function is a debug print function that will print out each element of the population including chromosome and fitness.

6.2.2 matrix

1. `std::vector <city_node> create_node_vector(const std::string &filename)`
   This function generates a vector of city nodes by processing the file given by `filename`.

2. `static std::vector <std::string> tokenise(const std::string &line)`
   This function tokenises a given line by splitting the line with a space delimiter and pushing into a token string vector.

3. `static int parse_dimension(const std::string &line)`
   This function parses an int from a given line as the dimension.

4. `static city_node *parse_node(const std::string &line)`
   This function returns a `city_node` pointer that points to a processed `city_node` object from the given line.

6.2.3 ga_frame

1. `int ga_main(std::vector <city_node> node_list, int population_size, unsigned long long int total_evaluations, int local_threshold, float elite_rate, float mutation_rate, float mutation_size, int warn_threshold)`
   This function is the main Genetic Algorithm driver and is the routine that governs the entire solving process.

2. `std::vector <int> create_genome(std::vector <int> main_genome)`
   This function creates a random solution by shuffling and returning a seed genome known as `main_genome`.

3. `unsigned long local_opt(std::vector <Individual> &population, std::vector <city_node> node_list, int max_calc)`
   This function takes a population address, node_list and maximum amount of fitness calculations to apply one generation of 2-opt local optimisation on the population and returns the amount of fitness evaluations done during the optimisation.

4. `std::vector <Individual> new_gen(std::vector <Individual> old_pop, std::vector <city_no node_list, float elite_rate, float mutation_rate, float mutation_size)`
   This function generates a new generation by running selection, crossover and mutation with percentage chances for each of the chromosomes on the population and returning the newly created population.

5. `std::vector <int> cross_chromosome(std::vector <int> chromo_1, std::vector <int> chromo`
   This function crosses two chromosomes and returns the newly made chromosome.

6. `std::vector <int> mutate_chromosome(std::vector <int> target_chromo, float mutation_rat float mutation_size)`
   This function potentially mutates a chromosome depending on a chance given as a parameter. A more definitive explanation on what mutation occurs is within the code comments.

6.2.4 Individual class functions

1. `Individual::Individual(std::vector <int> chromosome_init, std::vector <city_node> node_list)`
   This function is a class constructor that generates an object from the class template and the given variables.

2. `double Individual::calculate_fitness(std::vector <int> target_chromo, std::vector <city_node> node_list)`
   This function returns the total distance travelled according to the solution target_chromo.

3. `unsigned long Individual::local_2_opt(std::vector <city_node> node_list, int threshold)`
   This function returns the amount of fitness evaluations committed during the optimisation of a solution chromosome inside the object.

4. `std::vector <int> Individual::opt_2_swap(int i, int k)`
   This function returns the chromosome after it has undergone a opt_2 optimisation swap.

7 Code Snippet References

Code references are available as inline comments for whenever they are used.