Matúš Rusňák ID: 116286

UI – assignment 2c

In this assignment we were to create and implements a solution for the traveling salesman problem. This problem is about finding a shortest possible path for a traded that wants to visit every city. Cities are generated randomly with random coordinates as well. We can calculate the path between the cities with the help of Pythagorean theorem. One path must also be closed meaning from the last city we visited we must also return back to the first city. For this assignment I created 2 algorithms. Genetic algorithm and Tabu search algorithm. Output of solution must be best found order of cities and corresponding path cost.

1. Genetic algorithm

This solution is interesting because it is based on selection of genes that are stronger, random mutation and many other random variables. But it works really fast. The base premise is that you generate random chromones at the start of the program and then breed and mutate the best ones. I decided to follow know good practice that I read about which is as follows: Selected 64% of the population for breeding, 24% for random mutation and lastly add 8% of the population as random chromosomes to have good variance.

1.1 Path representation (chromosome)

I store my path in an object called chromosome. It has an attribute fitness which determines its path cost and attribute genes, which is basically a city order. It has a few methods, but they are just some helper functions to calculate distance between cities and such. This is how it looks in code.

```
class Chromosome:
    """
    class that stores genes(city order) and fitness (route cost) of genes
    """

    def __init__(self, city_order) -> None:
        self.genes = city_order
        self.fitness = self._calculate_cost()
```

1.2 Parents selection

For this algorithm I implemented 2 parental selections. You can choose between them an integer 1 or 2 based on which selection you want to use (1 = tournament, 2 = roulette selection)

1.2.1 Tournament selection

This selection works like it sounds. You select random chromosomes from the population and make them fight between each other (meaning you select the chromosomes that has the best fitness). Initial random selection is 1/10th of the whole population. You repeat this selection until you have enough selected parents and then you can start breeding between them. Code representation:

```
def _tournament_selection(self) -> List[Chromosome]:
    """
    method that selects parents for creation of offsprings
    this tournament chooses random 1/10th from population and selects best chromosome based on its cost
    this selection is run for until 64% of parents were selected from all population
    :return: selected parents that won the tournament
    """
    selected_parents = []
    while len(selected_parents) != int(consts.INITIAL_POPULATION * (0.64)):
        tournament = r.sample(self.population, int(consts.INITIAL_POPULATION * (0.1)))
        winner = min(tournament, key=attrgetter('fitness'))

        self.selection_winners[tuple(winner.return_city_order())] = True
        selected_parents.append(winner)
    return selected_parents
```

1.2.2 Roulette selection

Selection that is completely based on randomness as the name implies. But because of that there is a healthy variance in selected parents meaning new better solutions can be found. This selection works on a roulette principle, meaning that you place every chromosome on a roulette and the fitter it is (shorter path it has) the bigger its space on a roulette is. Basically, it has a greater chance of being chosen. You spin the wheel by firstly counting the cost of all chromosomes and then generating a random number between 0 and cost of all. When you have generated a number, you loop through every given chromosome and subtract that chromosome's cost until you have a number lower than 0. Once that happens the last chromosome u subtracted is a chosen winner of the roulette selection and is ready to for breeding. You repeat that until you have selected 64% of the total number of current population.

```
# calculating fitness value from worst node so that cost is reversed
for chromosome in self.population:
    fitness_value = worst_solution.fitness - chromosome.fitness
    sum_of_finesses += fitness_value

# edge case that happends when curent population is full of duplicate chromosomes
if sum_of_finesses == 0:
    return selected_parents

# for loop for spinning the wheel once wheel stops select parent
for i in range(int(consts.INITIAL_POPULATION * (0.64))):
    j = 0
    random_stop_point = r.randint(0, int(sum_of_finesses))
    while random_stop_point >= 0:
        stop_minus = worst_solution.fitness - self.population[j].fitness
        random_stop_point -= stop_minus
        if random_stop_point > 0:
            j += 1

    selected_parents.append(self.population[j])
    # remember selected parent
    self.selection_winners[tuple(self.population[j].return_city_order())] = True
return selected_parents
```

1.3 Breeding

Now that we have parents we can begin breeding. From 2 selected parents we want to create 2 offspring. That is done by random selecting a part of one parent, removing that selected part from second parent. We keep the rest of the second parent in the order that it is after removing the selected part and the randomly shuffling the selected part from first parent and appending to the second parent. We repeated this vice versa for seconds parent and we have successfully created 2 offsprings that can be added to our population. We do this until we breed all selected parents. Code:

```
offsprings = []
for i in range(0, int(len(parents) / 2)):
    # first offspring creation
    start = r.randint(0, len(parents[i].genes) - 1)
    end = r.randint(start + 1, len(parents[i].genes))
    offspring_end = parents[i].genes[start:end]
    parent2_copy = parents[i + 1].genes.copy()
    for city in offspring_end:
        parent2_copy.remove(city)
    offspring = parent2_copy + offspring_end
    offsprings.append(Chromosome(offspring))

    # second offspring creation
    start = r.randint(0, len(parents[i].genes) - 1)
    end = r.randint(start + 1, len(parents[i].genes))
    offspring_end = parents[i + 1].genes[start:end]
    parent1_copy = parents[i].genes.copy()
    for city in offspring_end:
        parent1_copy.remove(city)
    offspring = parent1_copy + offspring_end
    offsprings.append(Chromosome(offspring))
    i += 1
return offsprings
```

1.4 Mutating

After parental selection is done, we get a random sample of chromosomes that were not selected for breeding and mutate them. Mutation is done by choosing 2 random genes (cities) to be swapped in current chromosomes (city order). After we swap them, we reverse the order of cities between them in order to preserve the original path as much as possible. This only changes two city paths as we want it to. We repeat this until we mutated 24% of the current population. Code:

```
mutated = []
for chromie in chromosomes:
    start = r.randint(0, len(chromie.genes) - 1)
    end = r.randint(start + 1, len(chromie.genes))
    new_genes = chromie.genes.copy()
    new_genes[start:end] = reversed(new_genes[start:end])
    mutated.append(Chromosome(new_genes))
return mutated
```

1.5 Whole process

After program starts it generates random initial population.

```
def _generate_population(self) -> List[Chromosome]:
    """
    this method generates random populations based on number in consts
    :return: returns list of randomly generated chromosome objects
    """
    self.in_order_solution = Chromosome(self._read_cities())
    first_chomosome = self.in_order_solution.genes
    r.shuffle(first_chomosome)
    population = [Chromosome(first_chomosome)]
    for i in range(consts.INITIAL_POPULATION - 1):
        new_population = (population[i].genes.copy())
        r.shuffle(new_population)
        population.append(Chromosome(new_population))

    return population
```

When the first population is created, we run the algorithm. Meaning select parents based on selection method, mutate chromosomes and create random new chromosomes. From selected parents create offsprings append all new chromosomes to current population and lastly selected best half of current population. Repeat that for number of iterations that you set before the program is ran. At the end the program will output best found city order and path cost.

1.6 Genetic algorithm conclusion

Since this algorithm is random every time you run it and change the seed the solution will be different. There are few things however that may improve this algorithm and that is when choosing which chromosomes to mutate instead of choosing chromosomes that were not selected for breeding, we could choose randomly. Or implement another breeding option but other than that I strongly believe that this implementation is strong.

2. Tabu search

Second possible implementation for the traveling salesman problem is tabu search. This algorithm uses a so called tabu list. Which a list of forbidden moves that were used before. Moves are forbidden so we do not return to the same position and become infinitely stuck. Algorithm creates possible next moves, checks if they are tabu and if not select best one out of them. We repeat this process until a condition is satisfied and output the best-found city order with the lowest cost.

2.1 Neighborhood

I represented a one solution in an object called neighborhood. Neighborhood object stores city order and city order path cost. It has a few helper methods which calculate its path cost and method that returns possible moves. Possible moves are created by swapping 2 cities in city order and reversing other cities between them. That way we preserver the original city order as much as possible and create new city order that closely represents the old one.

```
class Neighbourhood:
    """
    class that handles neighbourhood functionalities
    """

    def __init__(self, list_of_cities, calc_cost=True):
        self.hood = list_of_cities
        if calc_cost:
            self.cost = self._calculate_cost()
        else:
            self.cost = 0
```

2.2 Tabu list

In order to store moves that are tabu I created an object called tabu list. This object stores tabu moves in a list that has a maximum size. Meaning if maximum size is overflowed, tabu list removes its last inserted element. Object also has an attribute hasp map which is used to quickly check if newly created city order is taboo or not. This means that program is a little bit more memory heavy, but it greatly increases time performance. Tabu list also must reduce the tabu moves durability every iteration, so it contains a few helper methods for that.

```
class TabuList:
    """
    class that stores tabu history
    """

    def __init__(self) -> None:
        self.tabu_history = [None for _ in range(consts.TABU_SIZE)]
        self.tabu_history_dict = {}
```

2.3 Tabu algorithm process

Tabu algorithm run process is simpler than the genetic algorithm. You add initial city order to the tabu list and then start a for loop until a condition is fulfilled. In the loop you reduce the iterations limits in the tabu list, create possible candidates from current selected neighborhood, select the best-found candidate and lastly add them to your tabu list. Code representation looks like this:

```
self.best_solution = Neighbourhood(self._read_cities())
self.tabu_list.insert_hood(self.best_solution)

for _ in range(consts.TABU_ITERATIONS):
    self.tabu_list.reduce_limit()
    possible_candidates = []

    for candidate in self.best_solution.return_candidates():
        if not self.tabu_list.check_if_tabu(candidate):
            candidate.calculate_cost()
            possible_candidates.append(candidate)

    self.best_solution = min(possible_candidates, key=attrgetter('cost'))
    self.tabu_list.insert_hood(self.best_solution)
```

3. Testing

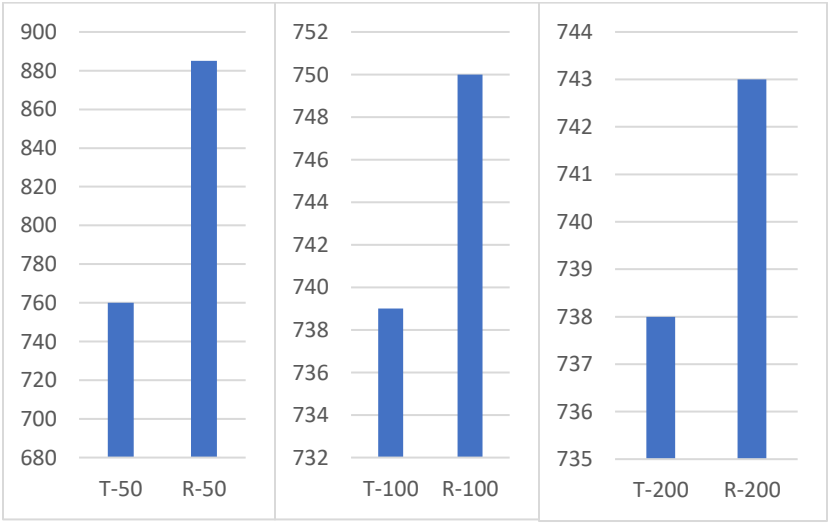
In these tests we will look at algorithm execution time and best-found solution cost while altering many conditions that heavily affect our algorithms like genetic iterations and tabu list size.

3.1 Genetic algorithm tests

While testing genetic algorithm we want to look at the difference between parental selections. We also look at how number of genetic iterations alter the result and lastly also how long it takes to execute the algorithm

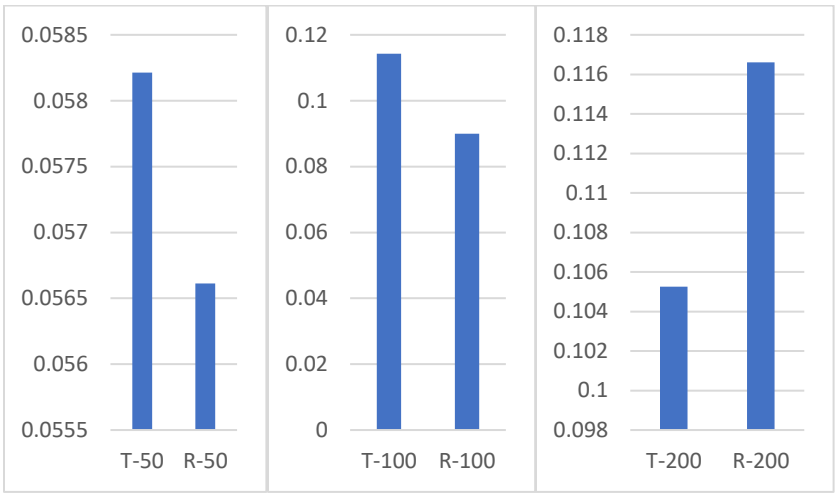
<u>Testing for 20 randomly generated cities</u>

Solution cost



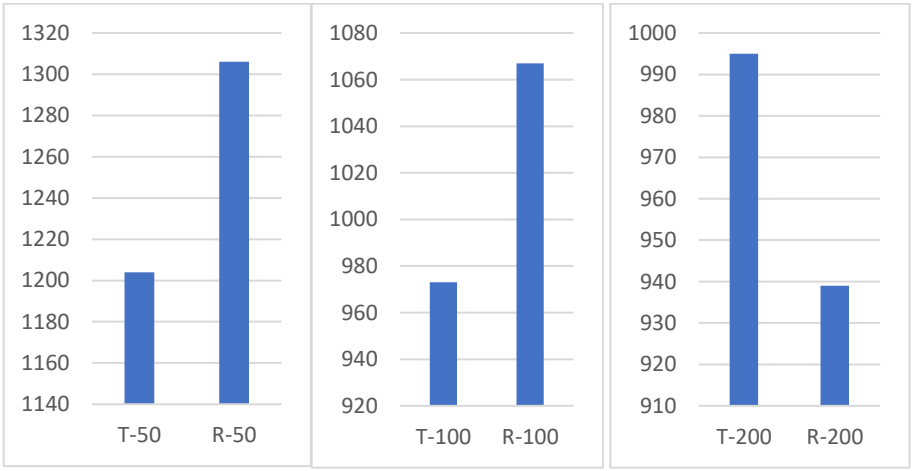
Matúš Rusňák ID: 116286

Execution time (in seconds)

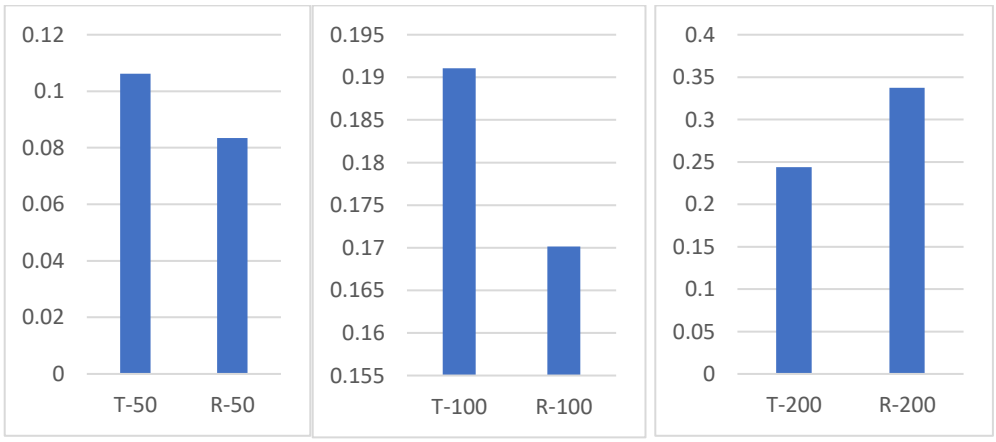


Testing for 30 randomly generated cities

Solution cost



Execution time (in seconds)



Values on x axis in graphs represent which parental selection was tested and with how many iterations. Example: T - 50 -> T = tournamet selection , 50 = 50 genetic iterations. Time was measured in seconds and solution cost could be represented as kilometers (my map size for random city generations was 200x200).

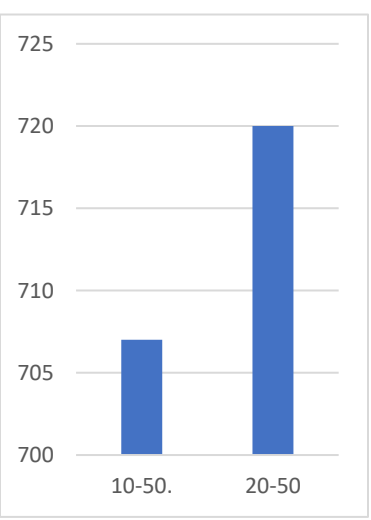
From these test results we can see that tournaments selection on average found better sollution than roulette selection. The results were not that much different but it is still noticable. Execution time on the other hand is reversed. Tournament selection on average looks to be slower than roulette selection. Keep in mind these results were generated on my machine with my randomly genereted cities. If you run them the results may be completely different. Another arguments to keep in mind is that genetic algorithm is random so with different seeds we could also get opposite results. But for these test I kept the same seed for every test.

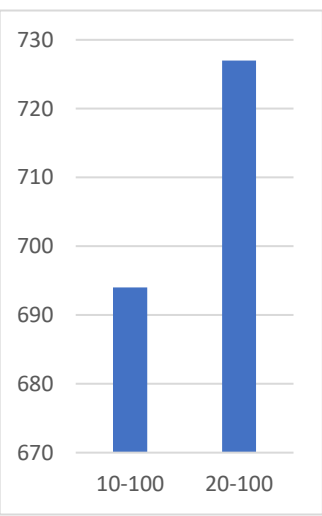
3.2 Tabu search tests

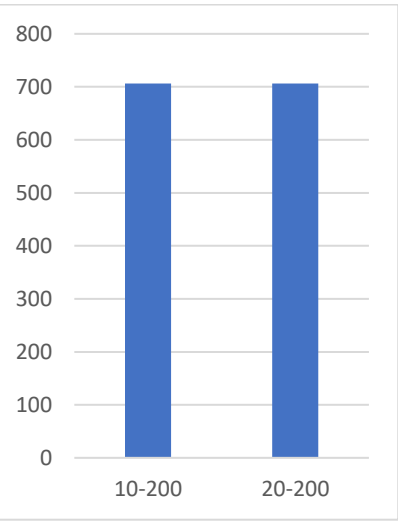
While testing tabu search algorithm we want to mainly look at different iterations, tabu list size and overall execution time. These conditions should affect the best solution found so lets look at my test results.

Testing for 20 randomly generated cities

Solution cost

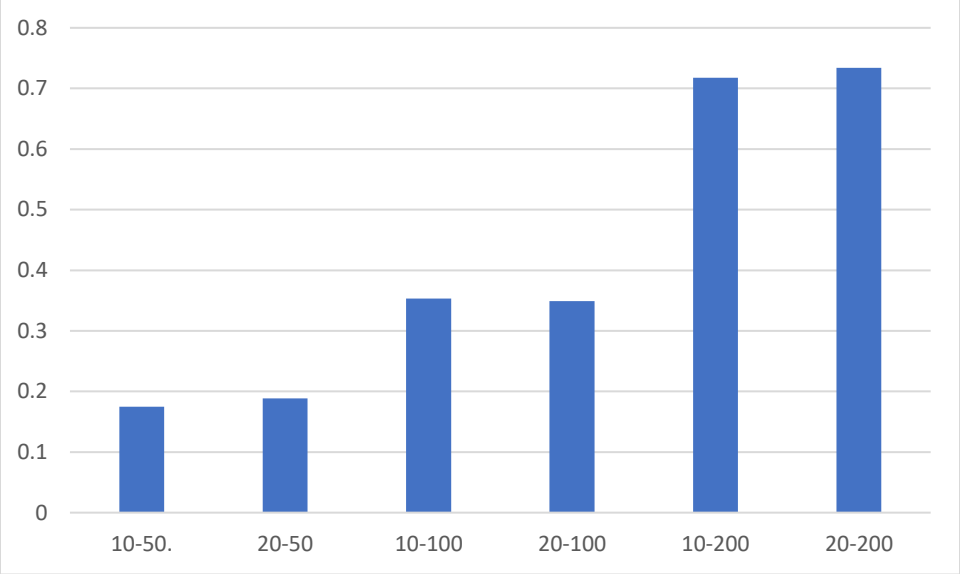






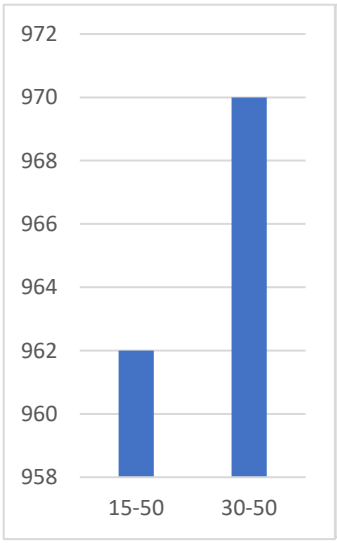
Matúš Rusňák ID: 116286

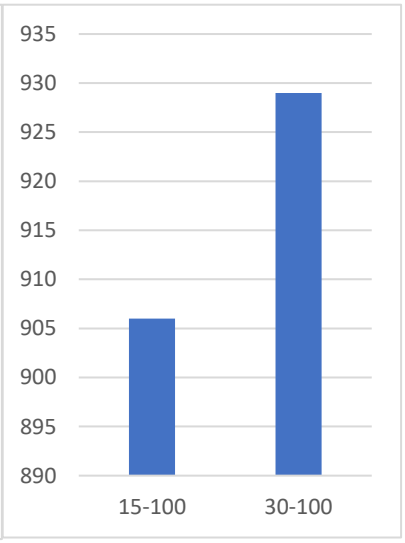
Time execution (in seconds)

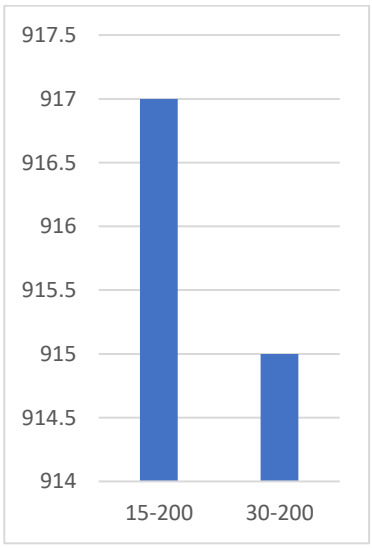


Testing for 30 randomly generated cities

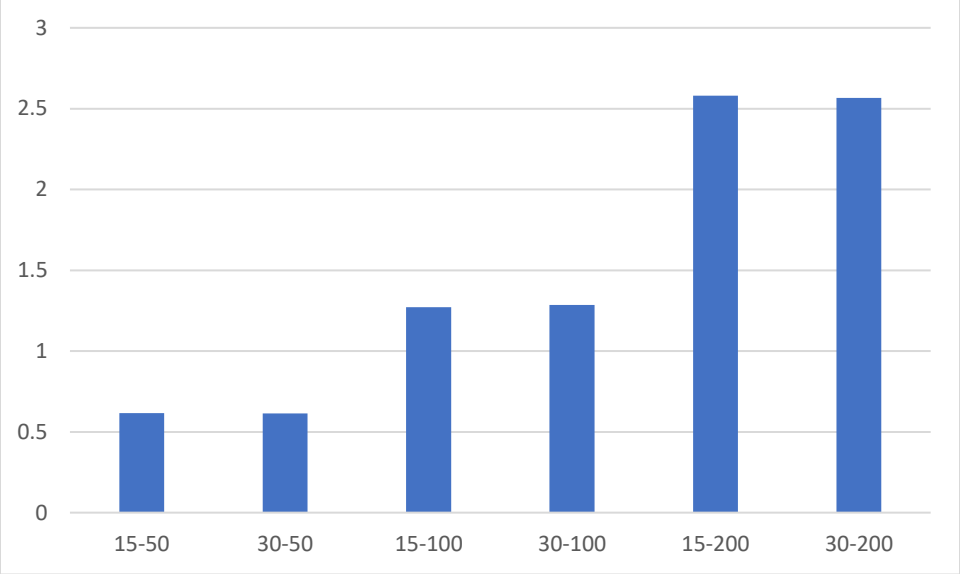
Solution cost







Time execution (in seconds)



Values on x axis in graphs represent which what was the tabu list size and for how many iterations did the algorithm run. Example: 10 - 50 -> 10 = tabu was limited to a size of 10 , 50 = algorithm ran 50 times. Time was measured in seconds and solution cost could be represented as kilometers (my map size for random city generations was 200x200).

From test results we can see that lowering the tabu size limit we actually found a better solution. This may be caused by the fact that even though the move was tabu from previous iterations It still lead to a better solution. If we look at execution time we can noticy that it is pretty much the same and that is because like I mentioned before I implemented a tabu list with a hashmap meaning that if we want to check if a move is tabu we get the asnwer pretty much instantly. This may cause however a greater need of memory space.

4. Conclusion

I am pretty confident that I created valid algorithms. They find an optimal path for every city order you give them. I really enjoyed creating the genetic algorithm as it was quite interesting to code selections and breeding for example. Although these implementations do not yield best possible results, they are certainly usable.

5. Usage

If you want to run this program open your terminal and change to project dir. Commands example:

python main.py –> will run these tests I created for this documentation

python main.py -s t -c y -> main has two parameters:

> '-s' -> which represents algorithm you want to run (t = tabu search, g1 = genetic algorithm with tournament selection, g2 = genetic algorithm with roulette selection)
>
> '-c' -> this parameter is used if you want to generate a new random city collection (y = True – create new random cities, t – which loads city date from test.txt, their best route is 426)

Project contains 'util' folder with consts.py file where you can change many different constants which will affect algorithm performance. For example, it contains seed for genetic algorithm, tabu list size, iterations for both algorithms and others.

6. Dependencies

No outside libraries or packages were used to develop this project. However, I developed it on PYTHON 3.10 so I recommended you use this version of python to avoid any problems or bugs.