ALGAV

3DE

1220514 – Liam Saliën

1220528 – Juuso Jalava

1220379 – Benjamin Moreira

1220459 – Patryk Pas

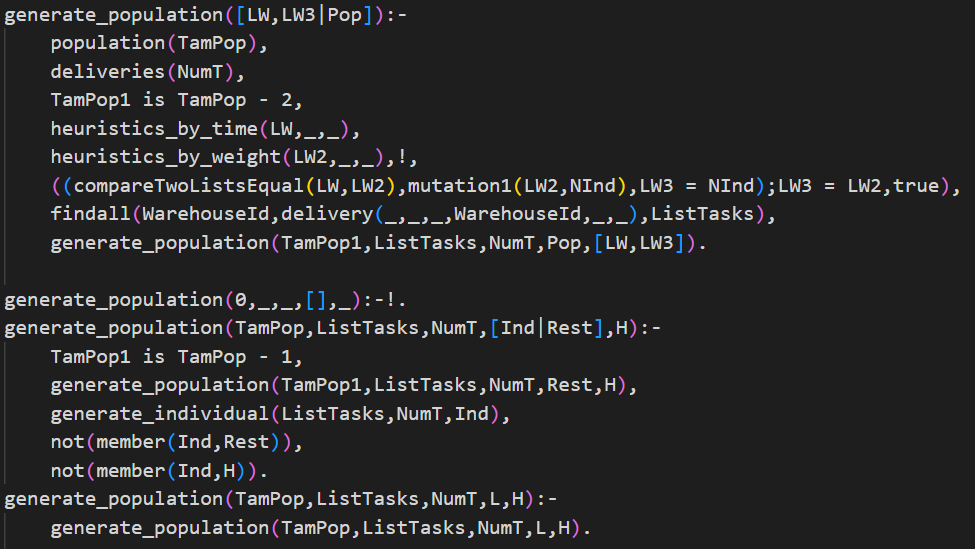
1220448 – Miika Kauppinen

# 1. Introduction

TBD

# 2. Creation of the initial population of the Genetic Algorithm (GA)

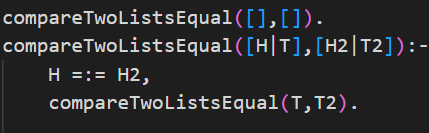
For this task, the example solution was adapted to our solution, and modified to include 2 different heuristics solution, and then randomly generated deliveries routes (from warehouse to warehouse):



Figure

The first generation population predicate (with one variable) will read the amount of deliveries that are in the system, and after that it will subtract the amount of individuals that are expected by 2, as it was required to include two solutions obtained by two different heuristics. The two heuristics used in our solution are by time, and by weight.

Furthermore it checks if the two solutions doesn’t equal each other (which uses a predicate that compares every single item in two lists).



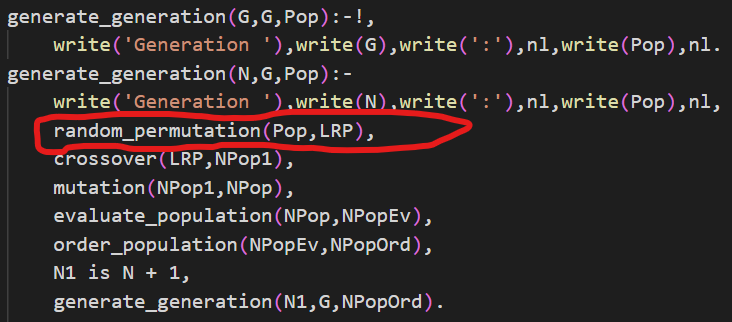
Figure

If this is the case, a mutation will be done on the second solution (obtained by the weight heuristic).

After this, for the rest of the individuals, another predicate will be called (and will include a list that already contains the two obtained solutions).

This predicate will be recalled until the TamPop is 0, which means we reached the amount of individuals we had to create. It will also check every single time if the generated solution is not already a part of the randomly generated deliveries routes list, and of the list that contains the two solutions obtained by the heuristics.

# 3. Random Crossover between individuals of the population



Figure

For this task the generate generation predicate was modified. The build-in predicate random\_permutation/2 is added after the current generation and their individuals are written to the screen.

Right before the written population will go through crossover and mutation etc, it is shifted by this build in predicate, so that the order will be different. Because of this, crossover doesn’t always happen between the first and second warehouseId, etc., etc.

*Example of this predicate in working:*

If we use random\_permutation on the following list:

*[[9,11,1,8,3]\*463.7758474576272,[11,8,1,3,9]\*486.5362288135594,[11,8,3,1,9]\*508.935593220339,[3,1,8,9,11]\*510.73241525423737,[11,3,9,8,1]\*537.482627118644]*

It might lead to this result:

*[[11, 8, 3, 1, 9]\*508.935593220339, [11, 8, 1, 3, 9]\*486.5362288135594, [9, 11, 1, 8, 3]\*463.7758474576272, [11, 3, 9, 8|...]\*537.482627118644, [3, 1, 8|...]\*510.73241525423737]*

This way our solution works more randomly, as well as the crossover takes place more randomly.

# 4. Selection of the new generation of the population

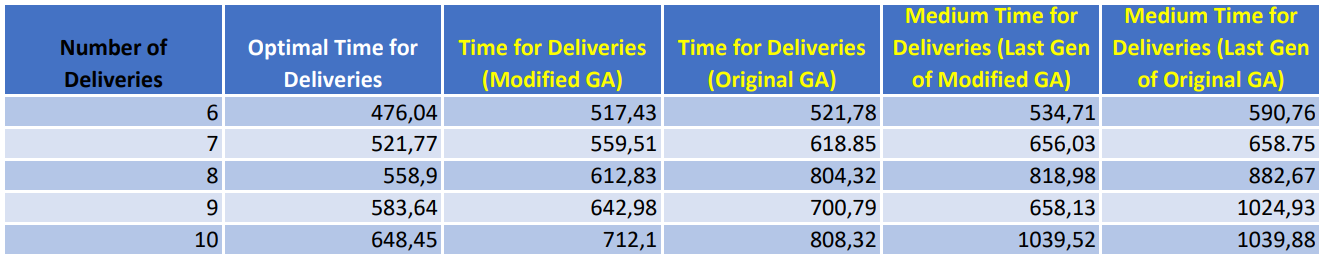
The selection of the new generation of the population considers the elements of the previous population and their descendants after crossover and mutation. It is important to consider that the mutation can change a descendant after crossover, potentially resulting in a better individual.

The two best individuals of each element (previous generation and descendants) are automatically included in the new generation. For the remaining elements, a tournament selection process is used to determine which individuals will be included in the new generation. In this process, any element can proceed, but individuals with better evaluations have a higher probability of being selected.

To implement this selection process, we can define a predicate that takes in the previous generation and the descendants and returns the new generation. This predicate can first identify the two best individuals from the previous generation and descendants, and then use the tournament selection process to determine the remaining elements of the new generation.

The tournament selection can be implemented using a random permutation, where we randomly select a group of individuals and compare their evaluations to determine the winner. The winner of the tournament is then added to the new generation, and the process is repeated until all the remaining elements have been selected.

# 5. Efficacy Analysis comparing the better individual of the created GA compared with the better from the initial version of the GA.



Figure

# 6. Parametrization of the ending condition of the AG

Text

Description automatically generated

Figure

The regular flow of a Genetic Algorithm flow (including crossover, mutation, etc.) was modified by including a stabilization check. If the populations of the latest generations (assembled by the append predicate) are all the same, we say it is stabilized (will be explained further below) and the predicate can stop. If not, it should do all the modifications and recall itself to generate new generations as long as it is not stabilized OR we didn’t reach the request amount of generations as mentioned/given during the initialization.

Text

Description automatically generated

Figure

We agreed that a population is “stabilized” when 10 generations after each other do have the same populations. This is a random number that was chosen, as it cannot be too small, and not too big, and stabilization doesn’t have only one possible solution/meaning (everyone can choose another definition of when something is “stable”). Is there a possibility new solutions will be created after this? Yes, but the obtained solution (when our program says it’s stable), is valid enough and won’t differ that much from further generated possibilities (as 10 times equal is already a lot).

These predicate firstly checks if there are already more than 10 generations, and if this is the case, it will reverse the generations as we only need the 10 latest ones. It will then check the latest population with the 9 populations that were generated before that one, and if all are equal, we can say it is stabilized.

# 7. Use of the GA to handle several trucks, representing in the same chromosome the deliveries of the several trucks.

TBD

# 8. Study of methods of Machine Learning

TBD

# 9. From a solution obtained from the GA it is envisaged to be able to allow dynamic changes

First, we must make deliveries dynamic so we can do changes to it (figure 7).



Figure . Predicate that makes delivery dynamic.

We have predicates for creating (figure 8), updating (figure 9), and deleting (figure 10) a delivery. Each predicate asks needed details from the user and carries out given task.

*Add\_delivery* asks user *Id, date, weight, destination warehouse, time to load and time to unload*. After that it uses *asserta* asserts a clause into the beginning of database. After that it writes “New delivery added” and ends the predicate.

Text

Description automatically generated

Figure . Predicate to add delivery.

*Change\_delivery* asks user Id and load. Then gets all other details of the delivery with given id, *retractall* deletes that delivery from the database we want to update and *asserta* creates an updated one. After that it writes “Delivery modified” and ends the predicate.

Text

Description automatically generated

Figure . Predicate to change delivery.

*Delete\_delivery* asks user Id of the delivery we want to delete and *retractall* deletes that delivery from the database. After that it writes “Delivery deleted” and end the predicate.

A picture containing timeline

Description automatically generated

Figure . Predicate to delete delivery.

To get the program to work with dynamic changes we need to make predicate which gives us the number of deliveries when we need (figure11).

Text

Description automatically generated

Figure . Predicate to get number of deliveries

We created predicate *generate\_dynamic* (figure 12), Which runs until *generate\_dynamic1* stops. At the end of *generate\_dynamic1, changes* predicate (figure 13) asks user to give 1, 2, 3 or 0. 0 or any else number or letter stops the predicate.

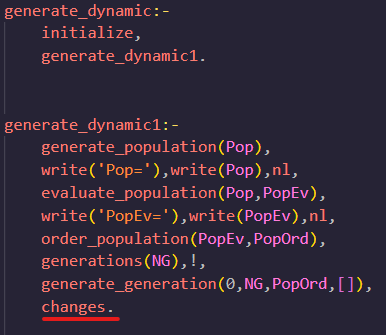


Figure . Generate predicate for dynamic changes

Text

Description automatically generated

Figure . Predicate which asks to type a number and do stuff according to it.

# 10. Conclusions

TBD