Katholieke
Universiteit
Leuven

**Department of
Computer Science**

SOLVING THE TRAVELLING SALESMAN PROBLEM

Using Advanced Genetic Algorithms

Jorge Arellano (r0779957)
Jesús García Ramírez (r0731792)

Academic year 2019–2020

Contents

1 Introduction

The travelling salesman problem (TSP), in which the shortest tour among a group of given cities must be found, is one of the best-known NP-hard problems. The most forward solution to this problem would be testing all the solutions. However, this resolution method will reach the solution in factorial time O(n!). Hence, there is a need for approaches that provide good-enough solutions in a reasonable time. Hence, genetics algorithms (GA) have been applied in numerous occasions to tackle this problem.

In this project, we have applied different genetic algorithms to try to solve this problem. First, the existing providing GA has been tuned. Second, we have developed our own approach in order to improve the results from the existing algorithm. Finally, we have tested our approach with benchmark instances, obtaining near-optimum results in reasonable time.

2 Existing genetic algorithm

In this section, we have studied the impact of different parameters of the existing GA in terms of performance. In order to study this impact, we have to define first what we consider a good performance for this problem. In this study, we have considered the TSP as a production problem. In these kind of problems, the best solving speeds and robustness of solutions are sought.

2.1 Datasets used

We have tested the existing genetic algorithm using four different datasets: rondrit016, rondrit048, rondrit070, rondrit127. We have chosen these datasets with two purposes. First, to address the performance of the approach across different datasets. Second, to evaluate the scalability of the parameters chosen. By using datasets with an increasing number of cities, we are also testing our approach for bigger search spaces. In this way, we can see how much the performance of our approach is degraded when we increase the search space.

2.2 Parameters considered

We have carried a two-stages process in order to test the existing GA.

- **First Stage**: In this stage we have used the parameters settings showed in table 1. We have decided to generate randomly the values of the parameters tuned within the intervals. The values are chosen randomly because of the results obtained in [9]. In this paper it is proven that randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid. We have fixed the number of individuals and generations to study the effect of the rest of parameters independently. The study of the influence of the number of individuals and generations is the goal of the second stage.
- **Second Stage**: This stage is conducted to assess the relationship between increasing the number of individuals and generations and the quality of the solutions. For doing so, we have chosen the three best candidates from the first stage, according to the performance criteria discussed in section 2.3. For each of these candidates, we have tested them for 150 generations and different number of individuals (20, 50, 80, 100, 120).

Parameters	Max Gen	No. Individuals	Elitism	Pr Crossover	Pr Mutation
Intervals	50	50	[0, 0.2]	[0.5, 1]	[0,0.5]
# Different Values	-	-	4	15	15

Table 1: Parameters used in First Stage

2.3 Performance criteria

Different performance metrics have been used for the different stages. In the first stage, we have address the quality of the solutions after the termination of each run. Since we have considered our problem to be a production problem, we have used average metrics to assess the performance of the approach. We have opted for Mean Best Average (MBF) and Mean Efficency (MEff) as performance metrics. The efficiency of each run

is defined below. $Ef(T)$ is the efficiency of a run at generation T. We have transformed our fitness into $\frac{1}{F_t}$ because in our problem higher fitness values correspond to worse solutions.

$$Efficiency = \frac{1}{NoIndividuals} \sum_{T=1}^{MaxGen} Ef(T) \qquad Ef(T) = \frac{1}{T} \max_{t=1,\ldots,T} \frac{1}{F_t}$$

To rank each of the parameter combinations tested, we have normalized the values of the correspondent MBF and MEff values across each of the datasets in the interval [0,1] where 1 means the best value of the correspondent metric in the dataset. Later, we sum the normalized values for the criteria across datasets for each parameter combination and we use this value (CritSum) as criteria to rank each parameter combination.

On the second stage, we have studied the temporal response of the runs for the different number of individuals in order to assess the convergence of each parameter setting of the GA.

2.4 Results

In this section, we present the three best parameter combinations for first stage test (table 2) and the scalability results for the best parameter combination (figure 1, and second and third combinations results are found in figures 15 and 16).

ELITIST	PR_CROSS	PR_MUT
0.140	0.566	0.338
0.140	0.631	0.190
0.136	0.566	0.190

Table 2: Best parameter combinations first stage

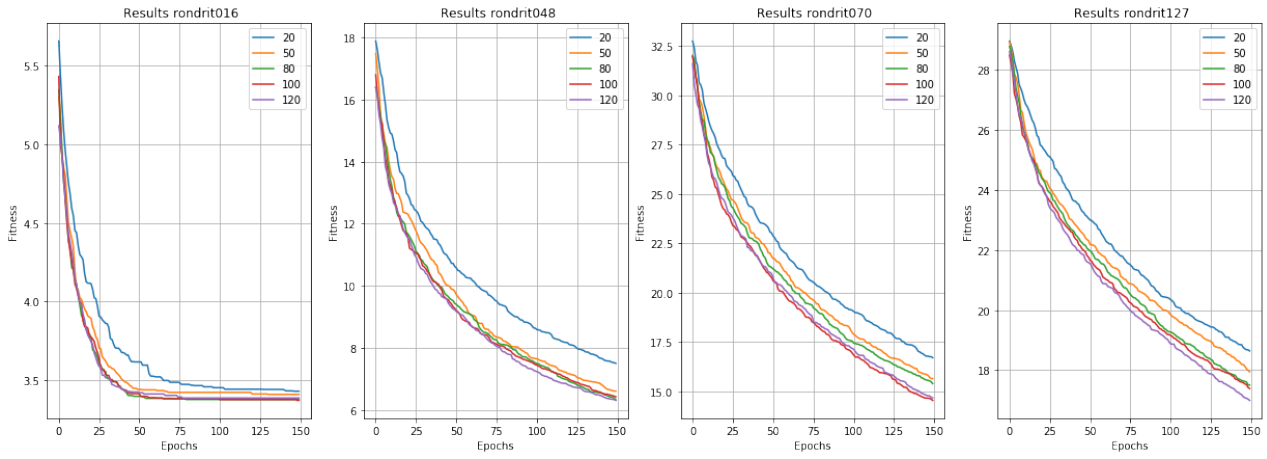

Figure 1: Scalability results best parameter combination (Elitist= 14 %, Pr Cross= 56.5 %, Pr Mut = 19 %)

2.5 Discussion

Next, a discussion of the effect of each of the parameters considered in our tests is presented.

Mutation rate The mutation that is implemented in the TSP problem is the exchange of two random cities. The choice here is the rate of mutation. If it is large, the algorithm will be more explorative: more random chromosomes will come up, which can be useful to escape from local optima. However, if the rate is small, more robust chromosomes (they will tend to be better) will be created, but we will be exploring less the search space. The best mutation rates depend on the problem. For example, in a problem with a high probability of falling into a local optimum, large mutations must be chosen in order to escape from it. After several tests, it has been proven that a good level of mutation rate is around 0.2-0.4.

Crossover rate

The crossover that the existing GA implements is the alternating edge crossover. Low rates perform the best. This result is clearly illustrated in Figure 9. We believe this operator is not too good. It sometimes randomly generates new edges, which is not desirable. Also, it does not use any knowledge about the problem. This may explain why low crossover rates give the best performance (50 %).

Elitism

Elitism is a parameter needed in order to make sure we don't discard the best individuals. However, the higher the elitism percentage the smaller the diversity of the population over the epochs. Results show that setting this parameter to zero decrease the performance of the algorithm. On the other hand, values in the interval [0.1-0.2] provide good results, since we get to conserve at least the best individual, and so the best fitness never falls in future epochs.

Population size and number of generations

Logically, the greater the number of generations, the greater the probability of having a good result. However, an increase in the number of generations is directly related with an extra computational cost. Hence, a good trade-off between finding good results and computational cost has to be sought. We can see that the existing GA has only fast convergence for the smallest dataset, whereas for the rest has a near linear decay. Therefore, drastically increasing the number of generations will not have a great impact for the smallest dataset. For the rest, a stopping criteria that takes into account the trade-off between computational cost and fitness has to be developed.

The effect of population size on the performance of the genetic algorithm is linked to population diversity and increased search space. Similar to the number of generations, higher values show improvement in the convergence and fitness of the population, but the quantifiable effect is not constant, since there is no linear relationship between the number of individuals and an improvement in fitness. In the problems considered the conclusion was reached that values between 50 and 100 individuals allow to obtain good results for the numbers of cities considered without a great negative impact on the performance of the algorithm.

3 Stopping criterion

Five stop criteria have been implemented and tested. In order to avoid early termination, the algorithm only checks the stop criteria once at least 25 generations have already been executed. As well, when evaluating the fitness, the stop criteria is executed on the best fitness of each generation.

3.1 Stop when the variance/mean of the fitness slightly changes in several generations

In this stop criterion the GA stops in case the fitness *variance/mean* of the last N (30) generations slightly changes. Slightly changes mean that if it drops under a threshold (0.01), the GA will terminate. The division *variance/mean* is used to normalize the fitness variance in each scenario.

3.2 Stop when an optimal fitness level is reached

In this case the GA stops when a previously known fitness level is reached. This is a very useful criterion if the optimal (or sub optimal) fitness level is known, in order to avoid unproductive executions. Moreover, this is a specially useful criteria when we are in the testing phase of our algorithm. If we are searching for the best hyper-parameter combination testing our GA using benchmark tests, we can use this stop criteria to easily discard those hyperparamaters combinations with an inadequate convergence according to this criteria.

3.3 Stop when a maximum number of generations is reached

In this case the GA stops when a maximum number of generations is reached. It is the original implemented stop criterion. This can be a useful criterion for avoiding too many executions.

3.4 Stop when the population diversity drops under a threshold

In this case the GA stops when the population diversity drops under a threshold. In the program, the GA stops when less than the 50 % of the individuals are different. This stop criteria is useful in order to terminate

the GA when local optima are reached due to the small diversity of the population. This could be avoided with a high mutation rate and performing techniques that preserve the population diversity.

3.5 Stop when the average rate of improvement drops under a threshold

In this case, when the improvement of the last N (30) generations drops under a limit (difference of 2 % between the best fitness of epochs $E - N$ and E) the GA stops. This case is very similar to the stop criteria 3.1, in which the GA stops when the fitness slightly improves in the last generations. However, this criteria has a more direct relationship with efficiency than criteria 3.1. This is because the threshold is directly imposed in the ratio of improvement. Also, this threshold could be tailored for each specific problem. This value can be seen as a ratio of how much computation power are we able to spend at what cost (improvement ratio).

3.6 Results and Discussion

In order to test the stopping criteria implemented, we have tested the existing GA using the best combination of parameters from the first section. We have tested the stopping criteria using the datasets from section 2.1. Finally, we have limited the possible number of generations to 1000 so it is not possible to reach infinite number of generations in the case of a poor stopping criteria.

The results from table 3 show similar behaviour of each of the stopping criteria for the three last datasets. This was an expected result due to the convergence responses shown in Figure 1. It is worth commenting the difference between the results of the second and fourth criteria.

Criteria 2 was tested using fitness values 10% smaller than the ones obtained in section 2.4. These values where an (almost) optimal value for the first dataset whereas they were far from any optima values for the rest of the datasets. Thus, the stopping criteria fails to give a reasonable stop for the first dataset. On the other hand, criteria 2 stops slightly after 50 generations (this was the limit of epochs used in section 2.4). Criteria 4 has the inverse behaviour of criteria 2 for these datasets. This is explained by the fact that the GA was already reaching an optima in the case of the first dataset. In this case, the individuals tend to converge to the fittest of the population. For the rest of the datasets, the individuals are not able to converge to the fittest because the fittest is changing (improving) with each generation, and seems to continue after 1000 epochs.

	Datasets							
	rondrit016		rondrit048		rondrit070		rondrit127	
Stop Criteria	Last gen	MBF	Last gen	MBF	Last gen	MBF	Last gen	MBF
Crit1	37	3.616	74	8.728	100	17.794	69	21.401
Crit2	1001	3.357	80	8.382	77	18.849	94	19.851
Crit3	150	3.406	150	6.687	150	15.306	150	18.049
Crit4	43	3.521	1001	4.556	1001	7.599	1001	9.222
Crit5	63	3.386	206	6.031	240	12.836	215	16.533

Table 3: Stop criteria results for different datasets

4 Path representation

In this section, we will discuss how a new representation leads to a completely new genetic algorithm. First, we will be discussing which representation we will be using, what considerations we must take into account and its advantages. Then, we implemented a series of operators (crossover and mutation), and performed fine tuning experiments to discover what operators work the best in this particular problem, as well as what were the other optimal parameters. Finally, we compared the performance of the new genetic algorithm with the given one.

4.1 Representation

We will be using a path representation. There are several reasons why we chose this one instead of any other (such as ordinal representation for example). The research effort on this representation is huge, so we can base our experiments in previous work. There are also tens of operators proposed. It is also a very intuitive representation, where the only constraint the genotype has to verify is that it is a permutation of the N cities (first N natural numbers). In case it is necessary, we also have to take into account that there are N genotypes

for the same phenotype, since the path is closed (ends where it started), and so we can choose as first element in our genotype N different cities.

4.2 Crossover operators

Researchers have proposed a huge amount of different crossover operators. Based on the course slides and some papers [2] and [3], we have implemented 5 different crossover operators. We chose the ones reported to have the best performance, along with a mix crossover, following our initial intuition that using different crossovers can help obtaining a wider variety of genotypes.

Sequential constructive crossover (SConst) Explained in the course slides, but implemented with slight differences. When selecting the next node of the child, if one parent's option is unfeasible (leads to a loop) and the other isn't, we just choose the feasible node as the next one. In the case none of them are feasible, we perform a greedy search to select the next node (will normally happen only when few nodes are left, so the search will not take that much computational effort).

Random sequential constructive crossover (SRand) Identical to the sequential constructive crossover, but the shortest distance is not deterministicly selected, but randomly selected, with a probability proportional to the inverse of the square distance. This may be useful to avoid premature convergence.

Edge recombination crossover (ERX) Implemented as explained in the slides. In case of tie, we select randomly.

Order crossover Implemented as explained in the slides.

Mix crossover Selects randomly between the four previous crossovers which one to apply each time it is performed.

We expect the results regarding the effectiveness of these crossover operators be coherent with the ones in [2] (Sequential Constructive is the best crossover a priori).

4.3 Mutation operators

Previous research also reveals some interesting mutation operators [4]. Once more, we chose those mutations reported to have the best performance, trying to avoid using similar mutations which will have too similar results. The mutation operators considered are:

Reverse sequence mutation Or reverse mutation, explained in the slides.

Partial shuffle mutation For each node, throw a p-coin and decide wether to mix it with another randomly chosen node or not (p = 1/N, where N is the number of cities, so we expect to shuffle one city with another).

Inverse 3 mutation Exchanges 3 randomly selected nodes.

Mix mutation Selects randomly between the three previous mutations which one to apply each time it is performed.

4.4 Parameter tuning

In this section, we have performed a number of tests in order to find the best hyperparameters combinations of our approach. We have used the same idea as in the first section for performing the tests. We have carried different stages in order to obtain the best finest relationship amongst the parameters.

- **First Stage**: We have tested the different combinations of Crossover/Mutation operators with the parameters shown in table 4. We have used the insights obtained from the results of the first section to choose the intervals. Thus, the elitism is kept fixed at 0.1 and we have lowered the mutation rate interval to [0,0.2]

- **Second Stage**: This stage has been performed after getting the best parameter relationships from the first stage. As we will show in the results, the best crossover/mutation operator combination is Sequential constructive crossover and Reverse sequence mutation. For this combination, it is shown that the best mutation rate is around 0.2. This stage is used to explore the effect of the remaining hyperparameter (Crossover rate). We have carried two sets of tests. First a coarse search using crossover rates in the interval [0.1,1]. Second, a fine search using crossover rates in the best interval found in the first sets of tests [0.8,1].
- **Third Stage**: We have perfomed a scalability test in this stage. We have tested the best hyperparameter combination found for 150 generations and same different number of individuals as in the first section.

Parameters	Max Gen	No. Individuals	Elitism	Pr Crossover	Pr Mutation
Intervals	50	100	0.1	[0.5,1]	[0,0.2]
# Different Values	-	-	-	15	15

Table 4: Hyperparameters intervals used for testing each Crossover/Mutation operator combination

4.5 Test results

The results of the first stage are shown in Figure 2. This figure reveals that the best crossover operator is Sequential constructive crossover, paired with Reverse sequence mutation. In the second stage, we explored the hyperparameters of crossover and mutation rates to find the best combination.We conclude with the results of first and second stage that the best hyperparameter combination is **Elitism = 10 %, Crossover rate = 90% and Mutation rate = 20%**. Finally, we compare in Figure 3 our tuned GA with the tuned GA from section 2. Both algorithms have been tested for 150 epochs and 50 individuals.

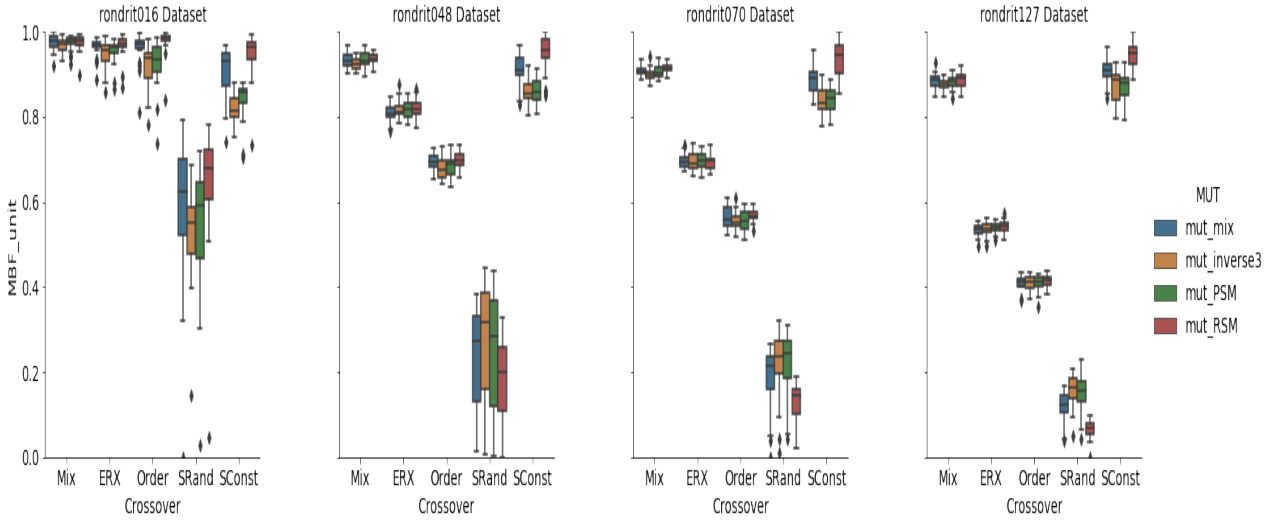

Figure 2: Relationship Mutation and Crossover operators with MBF.

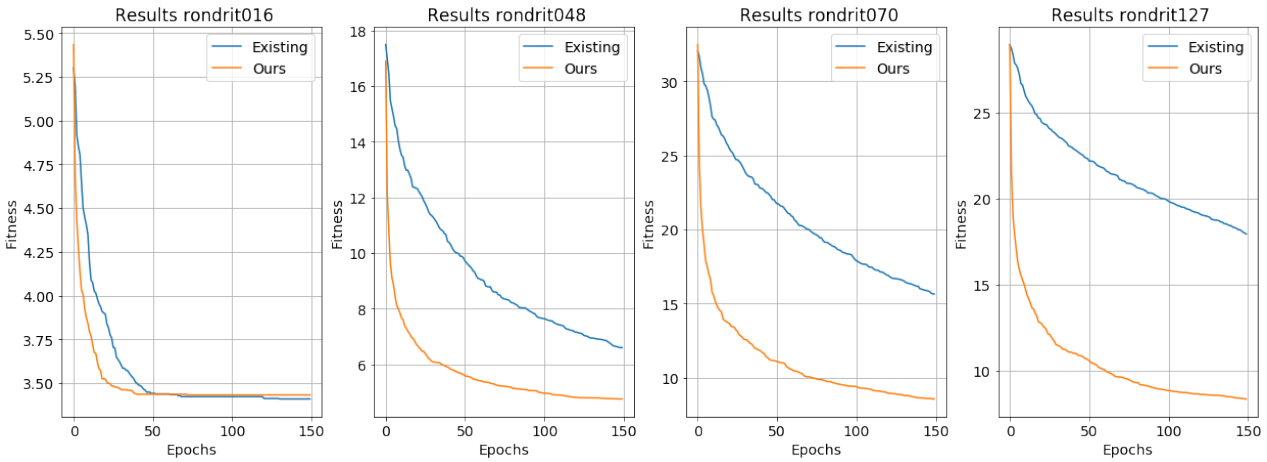

Figure 3: Comparison fitness evolution with best parameters from section 1 (Existing) and our best parameters (Ours)

4.6 Discussion

The fact that Sequential constructive crossover is the best one does not surprise us: previous work already found this out [3]. Mutation operators performance is more similar, but Reverse sequence operator is the best. This also matches the conclusion of [4].

As we can observe in Figure 3, especially with the bigger datasets, our algorithm is way better than the previous one. The main reason is that, in our operators, we are using problem knowledge (the distance between cities), while the previous algorithm did not. It is well known that using problem knowledge is always desirable since it will improve the performance of the genetic algorithm almost for sure. Another reason why we improved is that we performed an exhaustive hyperparameter optimization, trying different operators and rates of usage. However, we can not conclude that path representation is better (or worse) than adjacency representation. Several operators should be tried with adjacency representation to affirm this, which is out of the scope of this project.

5 Local Heuristic

Local optimisation consists of searching, maybe using specific problem knowledge, through the neighbours (we must define neighbourhood) of each chromosome in order to try to improve it. This will add computational effort to the algorithm, but it will make it converge more quickly to good fitness individuals. This search can be performed mainly anywhere in the course of the algorithm, but we perform it at the end of the generational loop.

5.1 Heuristics considered

After an exhaustive search in the web, we came up with 3 different heuristics for path representation, as well as with two variants. The heuristics were selected so that the computational effort did not seem very different between them, so when comparing the results we held a fair competition.

Path cut heuristic Described in [5]. The procedure consists of selecting an arc (i, j) and a node n not involved in the arc. The two neighbours of n are connected between them, leaving n unvisited. Then, the arc (i, j) is replaced by (i, n) and (n, j), closing the path.

Inverse two heuristic Proposed in [6], consists of choosing randomly a 4-subtour (a, b, c, d) and replacing it by (a, c, b, d).

Inverse n heuristic The natural extension of the previous heuristic, consists of choosing randomly a n-subtour $(a, b, z_1, ..., z_n - 4, c, d)$ and replacing it by $(a, c, Z_n - 4, ..., z_1, b, d)$.

Small greedy search Described in [7]. Consists of selecting a subtour randomly (of a random length between 2 and 20% of the tour length) and reorder it greedily.

Big greedy search Just the same as the previous heuristic, but longer paths are allowed (up to 50%).

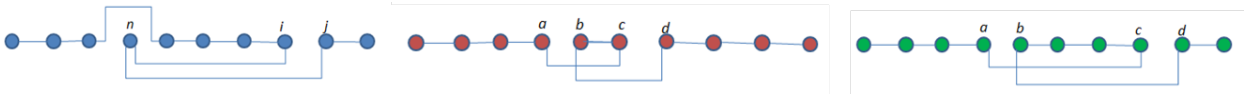

Figure 4: Sketch of Path cut heuristic (blue), Inverse two heuristic (red) and Inverse n heuristic (green)

The first three heuristics will operate in a greedy way. They will choose randomly one arc/subtour. If applying the heuristic improves the result, we finish. If it does not, we will retry with the next arc/subtour, until one improves the result or N attempts have taken place. For the last two, a unique attempt will take place. If the path does not improve, we leave as it was. The reason why we do not let these two heuristics make more attempts is because it would make the competition unfair. While the first three heuristics only have to recalculate 2-3 distances each time, the last two must perform a search. By limiting the number of attempts of the last two heuristics to one, we expect to balance the competition.

5.2 Experiments

We run the algorithm with the 5 different heuristics, as well as without heuristic. We will use the best configuration found in section 3. As well as plotting the evolution of the best fitness chromosome in each iteration (shown in figure 5), we will also count the number of times improvements take place along all the epochs (shown in section A.4) .

As we can see, the best two heuristics seem to be the last two, being the Large greedy search the best one by far. However, these results are misleading. We noticed will executing the code that using these two heuristics made the execution time increase considerably. It seems that the competition is not as fair as we expected. Especially when there are more cities, the search procedure take way longer than repeating N times the first three heuristics.

To solve this issue, we had three options:

- Making the competition fair by reducing the maximum length of the greedy search in the last two heuristics. This may be a good idea, but it is difficult to decide when the competition turns fair.
- Changing the effort measure so that heuristics are taken into account. Instead of counting fitness evaluations, we can count look-ups in the city distance matrix. That way, we can take evaluate heuristic efforts, including the greedy searches. If we did this, we would also have to take into account the efforts of mutations and recombinations that use distance between cities information. However, using such a simple basic operation can also mislead us, since we may overestimate the effort of looking up in such matrix, and underestimating other hidden operations that mutations, recombinations and local heuristics may perform.
- Measuring the CPU time. In an ideal world, this would be the best effort measure. However, results may not be reproducible, due to the hardware, the operating system or the network load.

We decided to go for the third option. We killed all the extra processes and disconnected the wifi. We also run a big number of experiments per heuristic, and did it in two different computers. By this, we tried to have the fairer environment to compare the CPU time of the different heuristics (see figure 6).

5.3 Results and Discussion

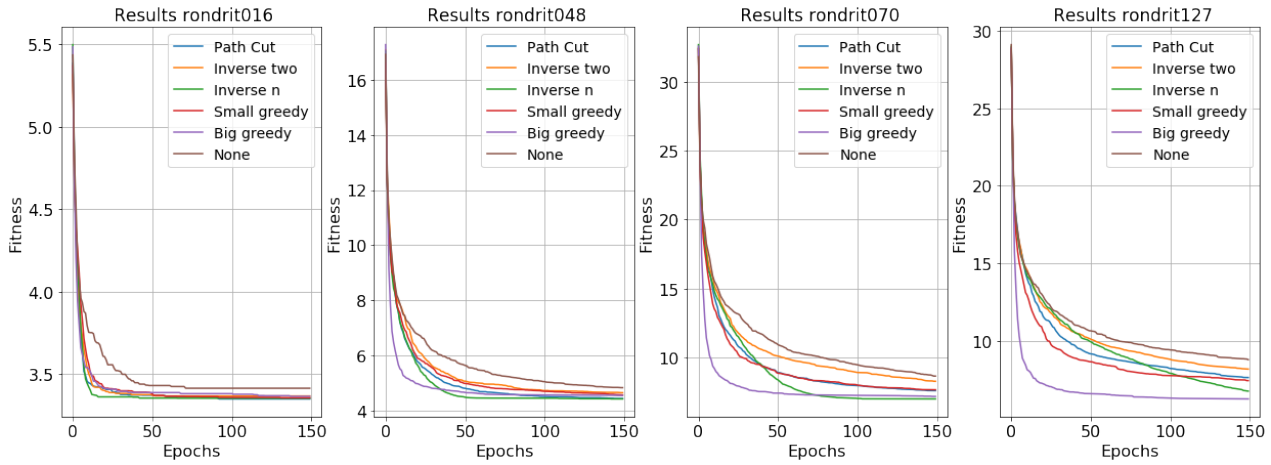

Figure 5: Fitness evolution over epochs for the proposed heuristics

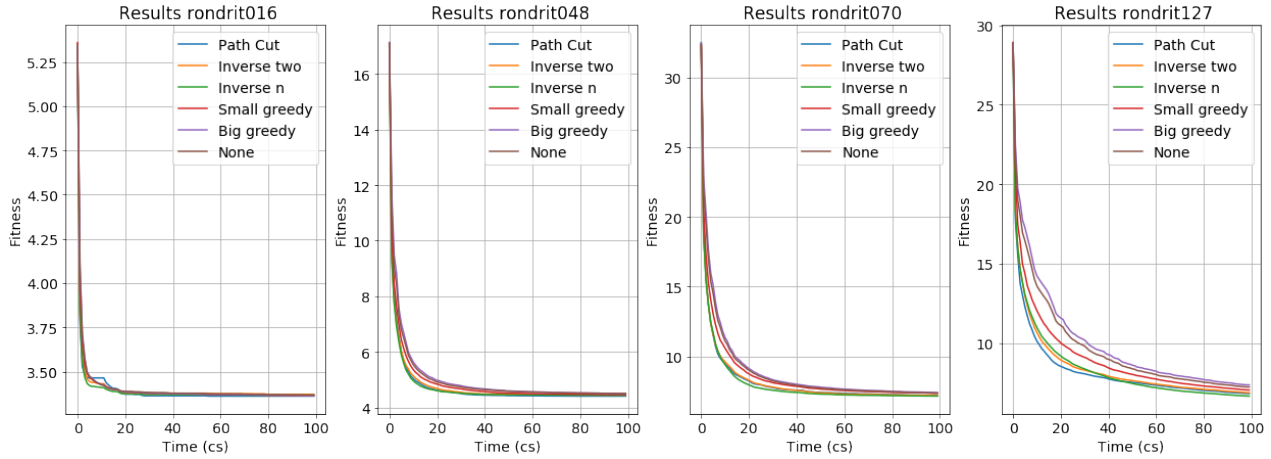

Figure 6: Fitness evolution over CPU time for the proposed heuristics

The main issue here was what effort measure was used. When we used fitness evaluations (without counting the additional effort of the heuristics), Big and Small Greedy heuristics were the best ones. However, we observed the computational effort of these was bigger (more CPU time) and we decided to measure the effort in CPU time. This changed drastically the results. Greedy heuristics now seem to be even worse than not using any heuristic at all.

However, we see that Inverse n heuristic, which performed quite well when the effort measure was the number of fitness evaluations (positioned between the top 3 heuristics), is the best one when the effort time is CPU time. Since greedy heuristics take too much CPU time, and Inverse n heuristic is the best in CPU time and in fitness evaluations without taking into account greedy heuristics, we conclude that it is the best heuristic for these datasets.

6 Benchmark Problems

We run the algorithm in four of the benchmark problems. The parameters used where those of section 4 with the heuristic Inverse n. The fact that there are two rows per dataset is the difference in targets: first rows target is $1.11 * optimum$ and the second rows target is $1.5 * optimum$. The columns are: the target, the average

fitness in the termination epoch, the number of epochs needed and the time in seconds needed. The computer properties are included in the appendix A.6.

Dataset	Target (km)	$F_{avg}(km)$	Epochs	Time (s)
xqf131	626.67	647.97	161	0.673
xqf131	846.00	974.409	80	0.3
bcl380	1801.11	1833.354	611	6.24225
bcl380	2431.50	2813.613	265	2.947
xql662	2792.22	2824.168	680	43.603
xql662	3769.50	4164.287	327	7.744
rbx711	3461.11	3506.11	688	47.854
rbx711	4672.50	4991.278	344	9.168

Table 5: Results test with benchmark datasets

7 Deterministic crowding

Premature convergence is a common issue of all genetic algorithms. When trying to find the global minimum, it is possible that all the chromosomes agglomerate around one or a few local minimum. This may occur in a number of scenarios: having a poor initialization, difficulties of mutation or local heuristic to escape from local minimum (due to the neighbourhood considered), or dominance of a local optimum chromosome over the population. However, with a good recombination and mutation operations, as well as mechanisms to avoid agglomeration of chromosomes, we can smooth this issue, granting a wider exploration of the space search. A popular way to deal with this is crowding, as explained in [1]. However, some problems were found with the classical approach, and a new one is suggested in [8]. That is why we will be implementing a refined version: deterministic crowding.

Since we want to maximize distance among paths (make them as different as possible between them by looking for the shortest path) we first must define a distance. Since we will be always comparing paths of the same number of cities, the simplest and most reasonable distance is $d(p_1, p_2) = N - ArcsInCommon(p_1, p_2)$. This way, two paths with no arcs in common have maximum distance, and two identical cities (may have shifted genotypes) have zero distance.

The algorithm can be found in both sources [1] and [8]. The procedure consists of determining, once two parents have recombined to produce offspring and these have mutated, which offspring relates more to which parent (pairing them), and selecting the chromosome (parent or offspring) with the best fitness to survive. This way, we maximize the distance between the survivors as well as selecting the fittest among the possibilities.

It is important to remark that the selection of parents must also be changed. The procedure consists of randomly pairing all the parents, having N/2 couples of different progenitors. This may not seem optimal, since the fittest chromosomes are not benefited. We tried to select the parents with the previous method (getting the N fittest among the parents and the offspring), but this had a poor performance: not only the efficiency decreased considerably, but also it was producing the opposite effect than expected (distance between chromosomes decreased). The reason why this happened will be covered in the discussion.

7.1 Experiments

We run the algorithm with the best parameters found in exercise 3, using deterministic crowding or not. We did not use any heuristic in these experiments. We plotted not only the fittest chromosome per epoch (Figure 7), but also the average intra-population distance (Figure 8).

As we expected, intra-population distance is way higher using crowding, and it more or less stabilizes at some point, while it has a descending trend when crowding is not used. However, crowding does not seem to improve performance of the algorithm.

7.2 Discussion

While coding crowding, we asked ourselves why parent selection should change. We arrived to the conclusion that is because, if we selected parents in the normal way, the same parent can be chosen more than once. If we select survivors with crowding, it is possible that we select more than once the same parent. If this parent is a good solution (or is a local minimum), it can multiply and overtake the population. This will mean a poor

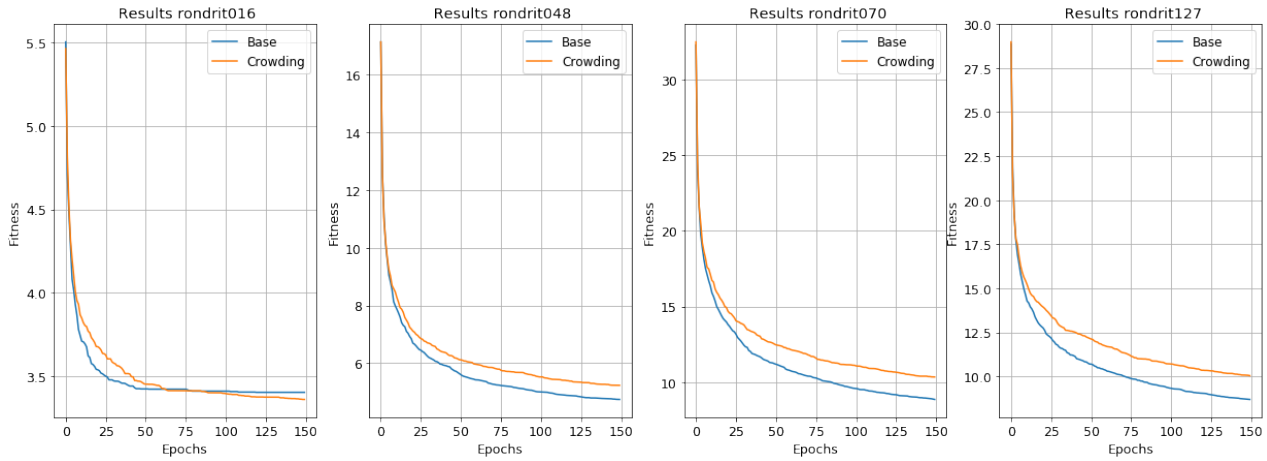

Figure 7: Comparison fitness evolution with and without crowding

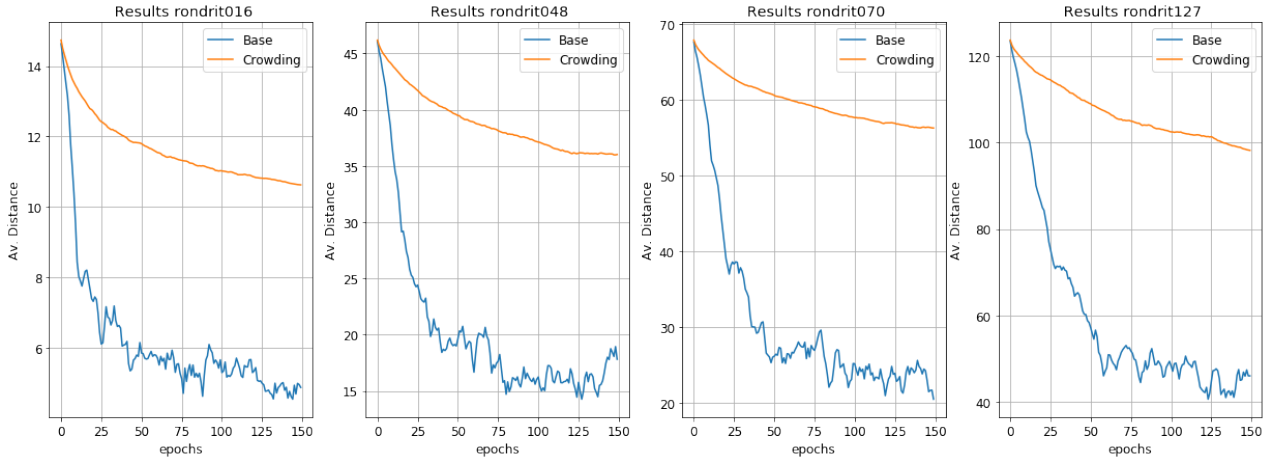

Figure 8: Comparison intra-population distance evolution with and without crowding

performance of the algorithm (is the minimum is not good enough) and the opposite effect we wanted in first place: similarities amongst the population will increase.

In Figure 8 we can see that crowding achieved its goal. The distance between chromosomes does not fall that quickly, compared to ranking selection. Actually, it remains nearly constant. However, crowding did not improve performance. This means that, at least for the small problems, it is better to perform a ranking selection.

However, crowding can be useful in other scenarios. First, if the problem has more local optimum, crowding might be a better choice since population diversity will help avoid early convergence. Also, if we have a problem where we want a wide variety of good solutions, deterministic crowding will guarantee a wider population diversity, thus finding more good solutions.

A Appendix

A.1 Results first stage of testing existing GA

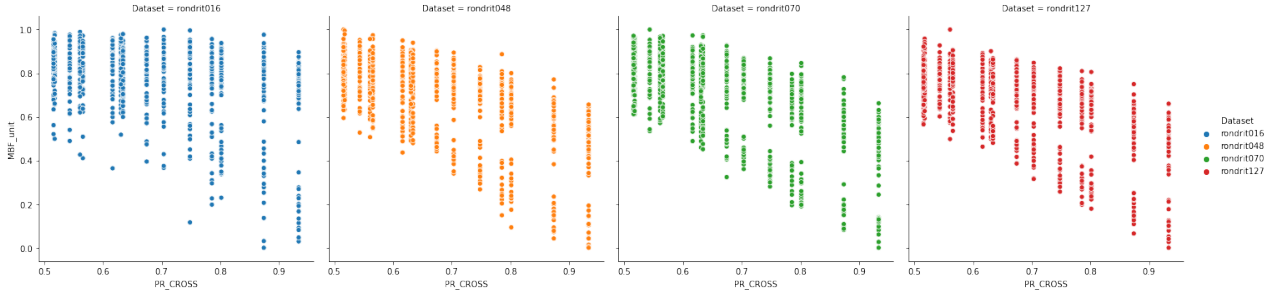

Figure 9: Relationship between PR Crossover and MBF

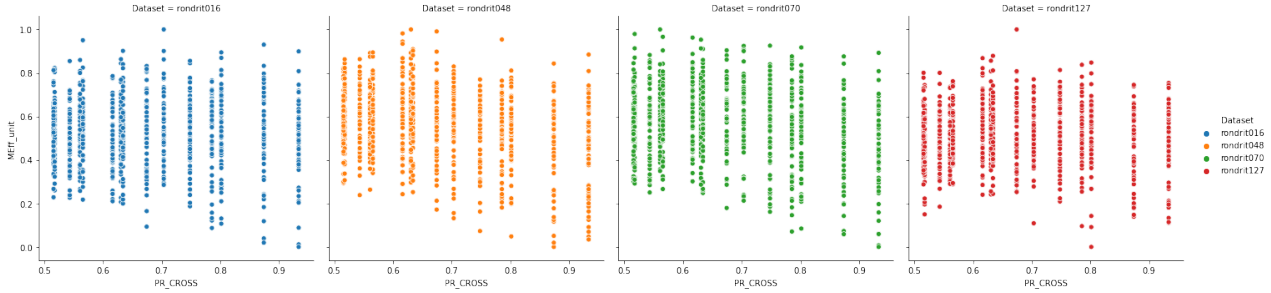

Figure 10: Relationship between PR Crossover and MEff

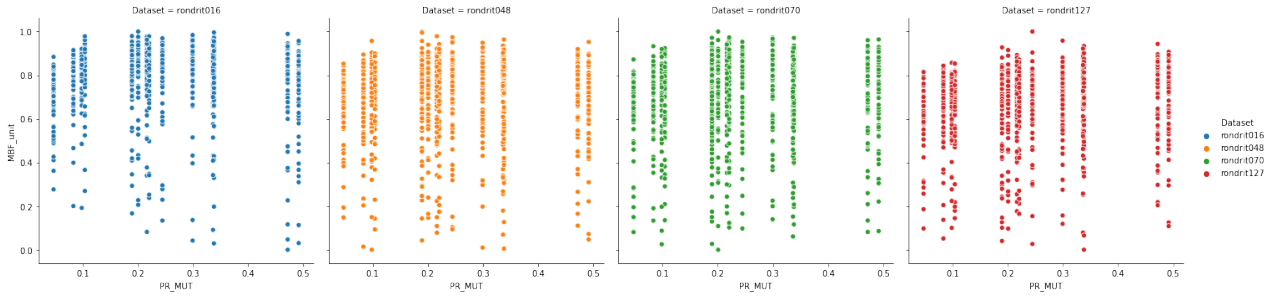

Figure 11: Relationship between PR Mutation and MBF

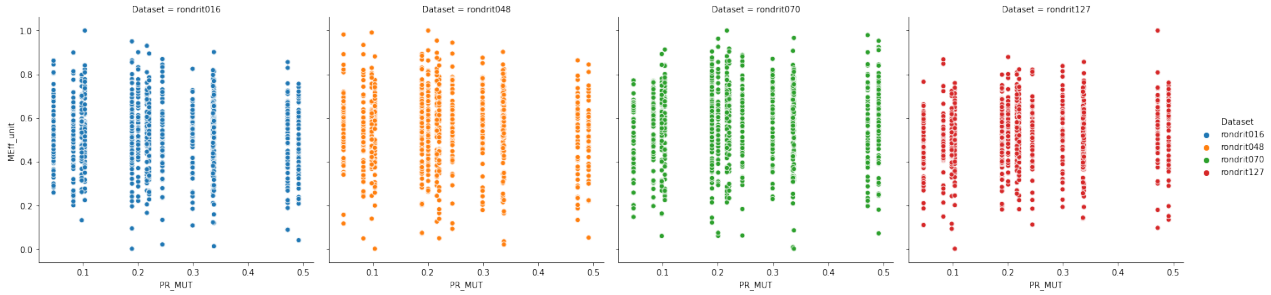

Figure 12: Relationship between PR Mutation and MEff

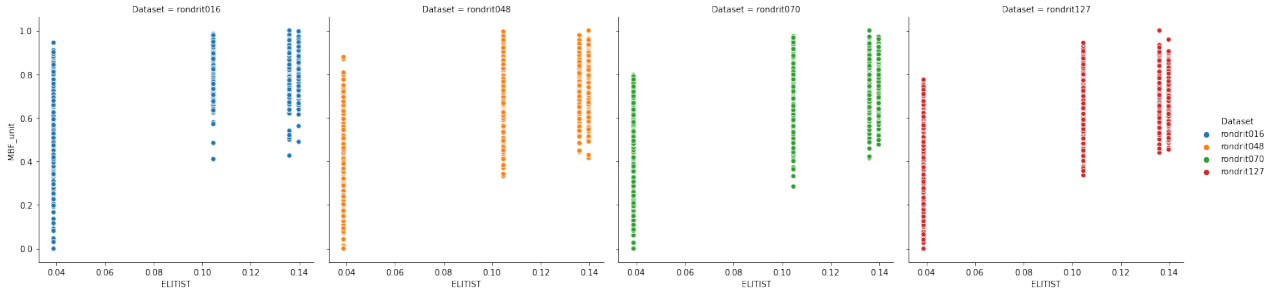

Figure 13: Relationship between Elitism and MBF

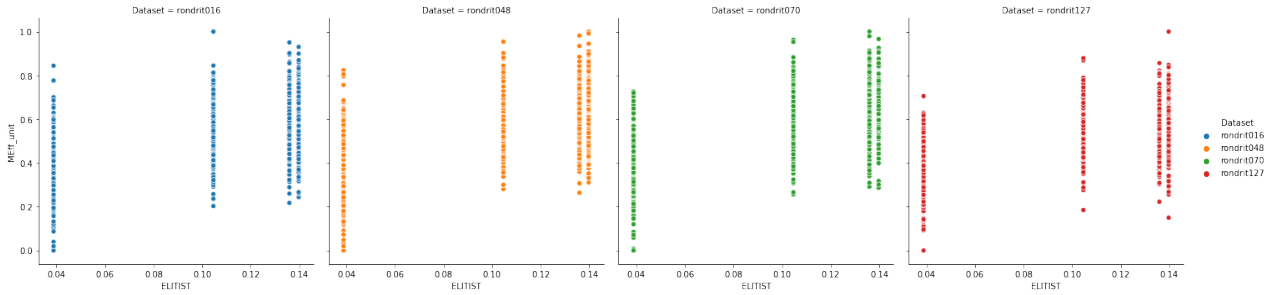

Figure 14: Relationship between Elitism and MEff

A.2 Scalability results existing GA

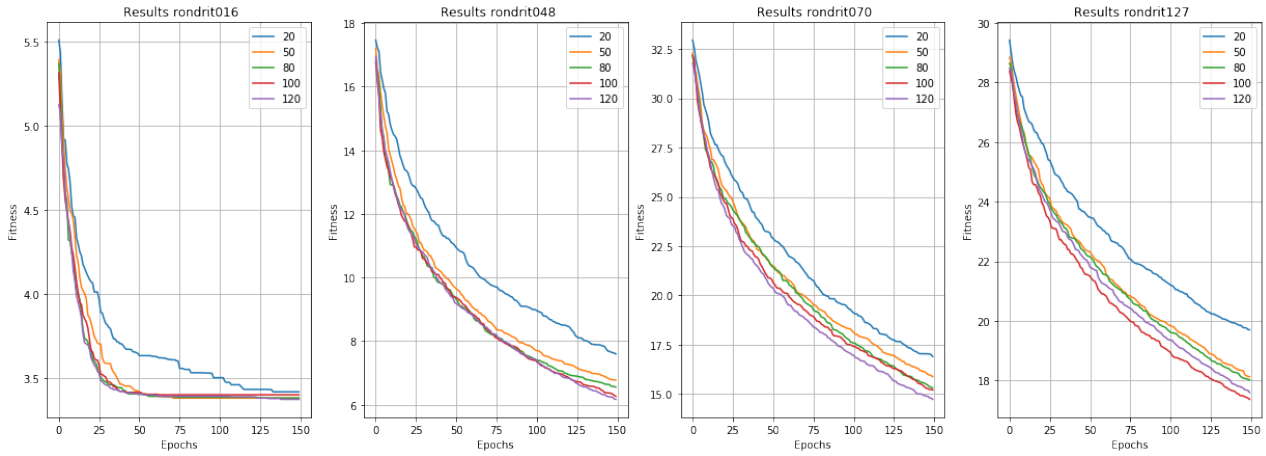

Figure 15: Scalability results second parameter combination

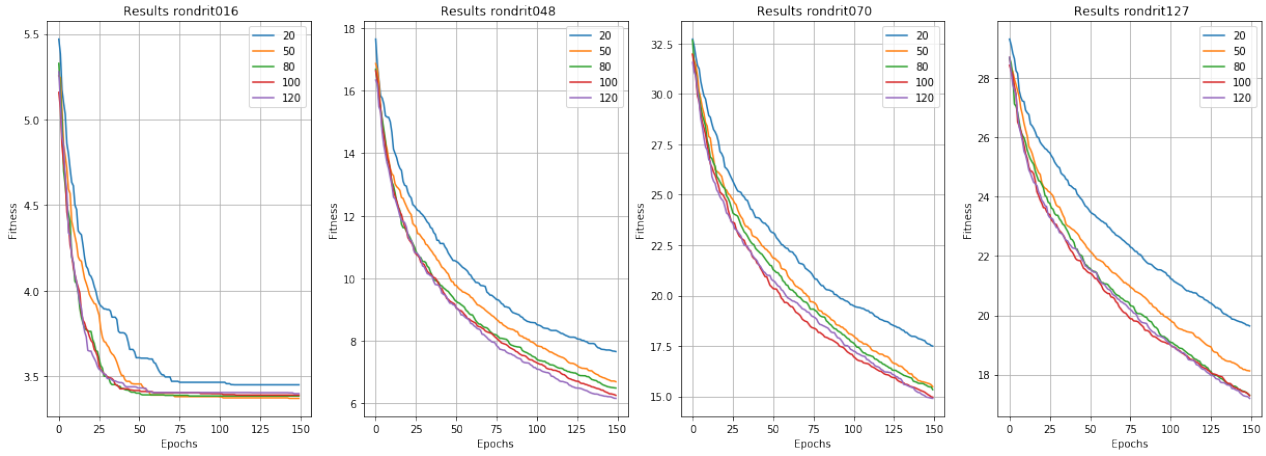

Figure 16: Scalability results third parameter combination

A.3 Results tuning hyperparameters Path Representation

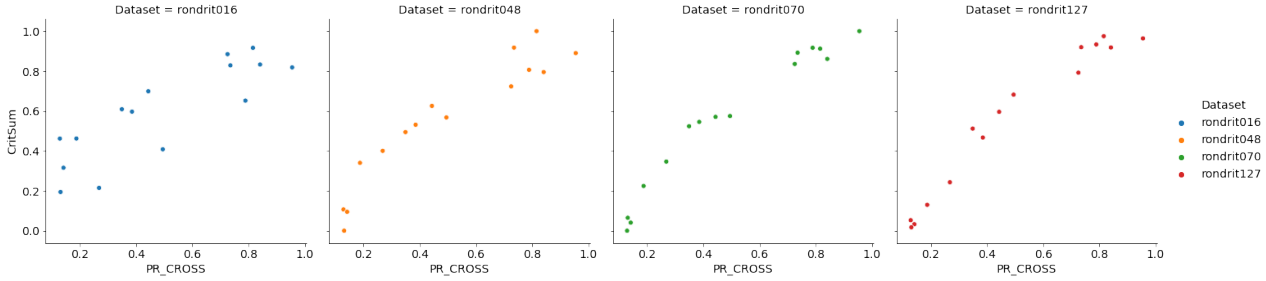

Figure 17: Relationship Crossover rate and MBF

A.4 Heuristics Results

Heuristic	MBF	Peak Best	NImprovements	Dataset
Path Cut	3.362	3.35	34096	rondrit016
Inverse two	3.378	3.35	33606	rondrit016
Inverse n	3.357	3.35	45543	rondrit016
Small greedy	3.388	3.35	26925	rondrit016
Big greedy	3.369	3.35	50161	rondrit016
None	3.4	3.35	0	rondrit016
Path Cut	4.466	4.33	58013	rondrit048
Inverse two	4.596	4.399	54954	rondrit048
Inverse n	4.443	4.327	48207	rondrit048
Small greedy	4.618	4.33	37535	rondrit048
Big greedy	4.528	4.362	39641	rondrit048
None	4.849	4.502	0	rondrit048
Path Cut	7.572	7.22	80367	rondrit070
Inverse two	8.339	7.286	68672	rondrit070
Inverse n	7.116	6.883	65323	rondrit070
Small greedy	7.823	7.274	41677	rondrit070
Big greedy	7.195	6.857	39763	rondrit070
None	8.832	8.116	0	rondrit070
Path Cut	7.435	6.845	127529	rondrit127
Inverse two	8.134	7.253	102070	rondrit127
Inverse n	6.689	6.254	138085	rondrit127
Small greedy	7.079	6.435	61287	rondrit127
Big greedy	6.362	6.027	57945	rondrit127
None	8.756	8.073	0	rondrit127

Table 6: Basic datasets results using different heuristics

A.5 Benchmark Results

Heuristics	MBF	Peak Best	Dataset
Path Cut	4387.78	3895.232	bcl380
Inverse two	4227.382	3494.901	bcl380
Inverse n	3169.562	2789.026	bcl380
Small greedy	2423.143	2161.142	bcl380
Big greedy	1844.475	1791.816	bcl380
None	4383.929	3847.624	bcl380
Path Cut	687.093	659.816	belgiumtour
Inverse two	699.778	677.713	belgiumtour
Inverse n	678.502	669.446	belgiumtour
Small greedy	701.794	679.241	belgiumtour
Big greedy	684.47	659.816	belgiumtour
None	705.841	659.816	belgiumtour
Path Cut	10826.76	9710.993	rbx711
Inverse two	9988.208	9395.093	rbx711
Inverse n	7109.542	5670.786	rbx711
Small greedy	4641.398	4287.251	rbx711
Big greedy	3599.984	3553.425	rbx711
None	10223.405	8793.975	rbx711
Path Cut	782.99	700.234	xqf131
Inverse two	848.286	790.639	xqf131
Inverse n	638.163	609.543	xqf131
Small greedy	694.772	662.161	xqf131
Big greedy	636.842	617.574	xqf131
None	909.757	800.174	xqf131
Path Cut	8726.947	7201.993	xql662
Inverse two	8022.303	7462.349	xql662
Inverse n	5752.307	5286.404	xql662
Small greedy	3887.445	3709.455	xql662
Big greedy	2922.898	2809.683	xql662
None	7949.38	6918.216	xql662

Table 7: Benchmark results using different heuristics

A.6 Computer properties

The properties of the computer used to run the tests are:
Processor: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80GHz
RAM: 16.0GB (15.9 GB usable)

References

[1] Introduction to Evolutionary Computing *A.E. Eiben, J.E. Smith*, 2015

[2] Comparison of eight evolutionary crossover operators for the vehicle routing problem *Krunoslav Puljicand Robert Manger*, 2011

[3] Genetic Algorithm for the Traveling Salesman Problem using Sequential Constructive Crossover Operator*Zakir H. Ahmed*, 2010

[4] Analyzing the Performance of Mutation Operators to Solve the Traveling Salesman Problem *Otman Abdoun, Chakir Tajani, Jaafar Abouchabaka*, 2012

[5] Solving TSP with Novel Local Search Heuristic Genetic Algorithms *Jianxin Zhang, Chaonan Tong*

[6] An improved genetic algorithm with a local optimization strategy and an extra mutation level for solving traveling saleman problem *Keivan Borna, Vahid Haji Hashemi*, 2014

[7] A Greedy-Genetic Local-Search Heuristic for the Traveling Salesman Problem *Mohammad Harun Rashid, Miguel A. Mosteiro*, 2017

[8] Crowding and Preselection Revisited *Samir W. Mahfoud*, 1992

[9] Random Search for Hyper-Parameter Optimization *Bergstra J., Bengio J.*, 2012