UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

COS790 Assignment 3

Generation Constructive Hyper-Heuristics

Student Number: u15024522

1 Introduction

This assignment involves implementation of a generation constructive hyper-heuristics to create heuristics for the symmetric and asymmetric Travel- ling Salesman Problem (TSP). More specifically, the toolset provided by ECJ [1] - a Java toolkit for evolutionary algorithm hyper-heuristics - was used to construct a genetic programming approach to generate and evolve a population of parse trees for prioritizing routes for travelling salesman problems in the Travelling Salesman Problem Library [2]. All of the code that was used to employ the genetic algorithm can be found on GitHub at `https://github.com/marcus-bornman/cos_790_assignment_3`. The repository also contains all information necessary to reproduce the results that are discussed in this assignment.

2 Representation

2.1 Nodes

The possible nodes for each individual, or parse tree, comprise nodes from a function set and terminal set. As per the implementation of the EvoHyp [1] toolset, the function set - used for possible internal nodes - contains simple addition, subtraction, multiplication and division arithmetic operators. In the case of the division operator, a value of 1 is returned if the denominator is 0. In addition, arithmetic rules were used so the function set also comprises an *ifelse* operator and relational operators ($<$, $>$, $<=$, $>=$, $==$ and $!=$).

In terms of the terminal set, there are 3 possible nodes - A, C and F - which represent attributes of a specific city, n, in the TSP instance. A represents the average distance from n to each of its neighbors. C represents the distance between n and its closest neighbor. F represents the distance between n and its furthest neighbor. These end nodes, as well as the aforementioned internal nodes, are described in more detail in table 1.

Symbol	**Arity**	**Description**
+	2	Adds its 2 children and returns the result
-	2	Subtracts the first child from the second child and returns the result
*	2	Multiplies its 2 children and returns the result
/	2	Divides the 1st child by the 2nd child and returns the result; or, returns 1 if the 2nd child is 0
ifelse	3	If the 1st child evaluates to true the 2nd child is returned; otherwise, the 3rd child is returned
<	2	True if the 1st child is less than the 2nd child; otherwise, evaluates to false
>	2	True if the 1st child is greater than the 2nd child; otherwise, evaluates to false
<=	2	True if the 1st child is less than or equal to the 2nd child; otherwise, evaluates to false
>=	2	True if the 1st child is greater than or equal to the 2nd child; otherwise, evaluates to false
==	2	True if the 1st child is equal to the 2nd child; otherwise, evaluates to false
!=	2	True if the 1st child is not equal to the 2nd child; otherwise, evaluates to false
A	0	Returns the average distance from a specific city in the TSP and each of its neighbors
C	0	Returns the distance from a specific city in the TSP and its closest neighbor
F	0	Returns the distance from a specific city in the TSP and its furthest neighbor

Table 1: Function and Terminal Set

2.2 Individuals

Each chromosome within a population consists of a parse tree for calculating a priority value for each city in the TSP. To clarify, when evaluating the fitness of an individual, the parse tree will be used to calculate a priority value for all cities in the TSP; then, the cities will be visited in order of these priority values and the total distance determined. Given the possible nodes identified in table 1, an example of one such parse tree is depicted in figure 1.

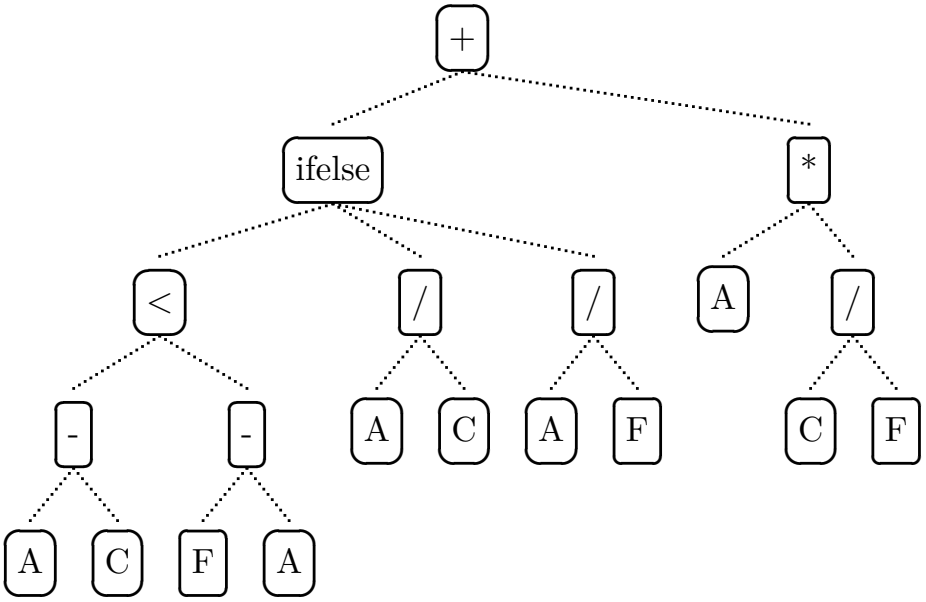

Figure 1: Example Individual

2.3 Populations

Of course, a population consists of a collection of the individuals that have been described. The genetic algorithm applied only made use of a single population consisting of 256 individuals. This means that only a single population was evolved and cross-population breeding was not applied.

3 Initial Population Generation

As implemented using EvoHyp [1], the initial population was generated using the *grow* method proposed by Koza [3]. At the maximum depth the elements from the terminal set are selected; but, at other levels, elements are selected from both the function and terminal sets, whilst abiding by constraints such that the child of an *ifelse* node must be one of the conditional operator nodes. The maximum tree depth allowed during the initial population generation was a tree depth of 4.

4 Fitness Function

To determine the fitness of individuals, we use the distance travelled between cities in order of the priorities calculated by the individual parse tree. To be more specific, for each city the priority is determined using the parse tree of the individual in question; then, after the priority for all cities have been calculated, the cities are ordered by their priorities to provide a route - or solution - for the TSP. Of course, using this calculation for the fitness of an individual means an individual with a lower fitness value is deemed better than one with a higher fitness value. Algorithm 1 depicts the fitness function in its entirety.

Algorithm 1: Fitness Function

```
foreach city in cities do
    city.priority = individual.decisionTree.calcPriority(city);
end
cities.orderBy(city.priority);
return calcDistance(cities);
```

5 Selection Method

Tournament selection [4] was used as the selection method for all genetic operators. This entails selecting T individuals entirely at random from the existing population. Then, from this set of individuals (ie. the *tournament*) the fittest individual is selected. After some initial test runs the size, T, of the tournament used was 16. On average, this tournament size produced the best balance between genetic diversity and convergence. In the cases where genetic operators required more than one individual from the existing population, the tournament selection method was simply reapplied to select each individual.

6 Genetic Operators

As per the EvoHyp toolkit [1], the genetic operators used included reproduction, crossover and mutation. Detailed by Koza and Poli [5], reproduction entails copying the selected individual program to the new population as is; crossover entails randomly selecting a crossover point in each of 2 parents and swapping the subtrees rooted at these points; and, mutation entails randomly selecting a point in an individual and replacing the subtree rooted at that point with a randomly created subtree.

Furthermore, every operator is associated with a probability which determines the likelihood that it will be used to select the next individual for the new population. The probability for crossover was selected as 0.8, the probability for mutation was selected as 0.1 and the probability for reproduction was also selected as 0.1. In addition, a maximum tree depth of 20 was applied as a constraint to offspring; and, the maximum mutation depth for subtrees created by the mutation operator was 4. With the selection methods and genetic operators having been identified, figure 2 depicts the entire breeding pipeline for generating a new population from an existing population.

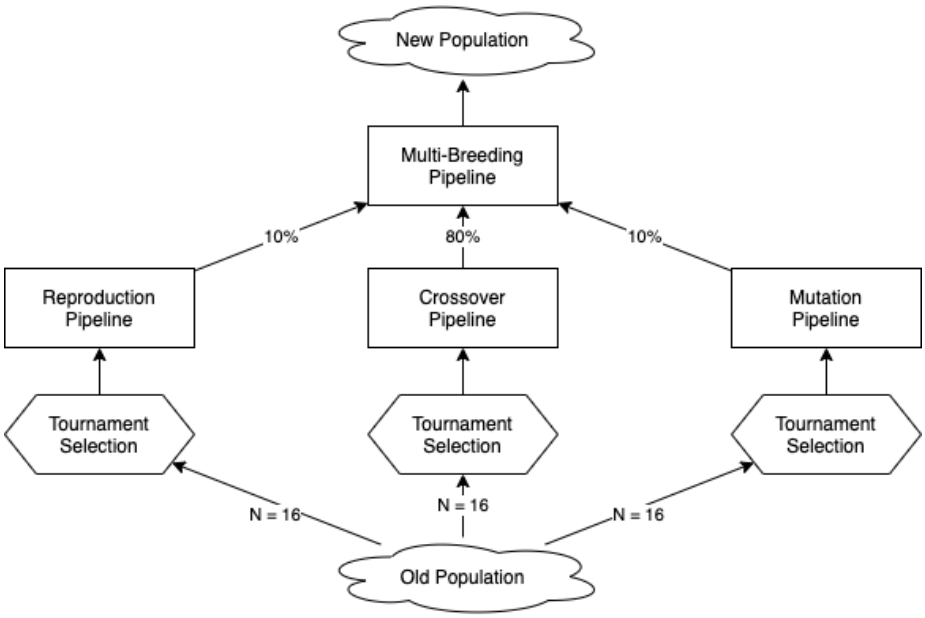

Figure 2: Breeding Pipeline

7 Experimental Setup

Firstly, the parameter values for the genetic programming approach have been mentioned throughout the preceding sections. However, for reference, they are summarized with the rest of the parameters below:

- Population Size = 256
- Tournament Size = 16
- Crossover Rate = 0.8
- Mutation Rate = 0.1
- Reproduction Rate = 0.1
- Max Initial Depth = 4
- Max offspring Depth = 20
- Max Mutation Depth = 4
- Num. Generations = 128

In addition, as has been mentioned, instances in the Travelling Salesman Problem Library [2] were used for experimentation. The specific instances used comprised small, medium and large problems of which 5 were symmetric and 5 were asymmetric problems. The symmetric problems were *ch150*, *eil51*, *eil101*, *rat195* and *rd400*. The symmetric problems were *ftv170*, *rbg323*, *rbg358*, *rbg403* and *rbg443*.

Finally, the machine used for development and testing purposes was a 2015 15-inch Macbook Pro with a 2.5 GHz Quad-Core Intel Core i7 processor, 16GB of memory and 500GB of storage. At the time of experimentation the machine had a battery cycle count of 684.

8 Results

To run tests a script was written which would run the algorithm on each instance 10 times, recording the solutions, runtimes and fitness values for each of the runs. The script is available in the github repository for this project (available at `https://github.com/marcus-bornman/cos_790_assignment_3`). Table 2 summarizes the results for each of the problem instances over the 10 runs.

Problem	**Opt. Distance**	**Avg. Runtime (milliseconds)**	**Avg. Distance**	**Best Distance**
ch150	6528	26178.2	35034.7	33169
eil51	426	3677.0	1140.5	1059
eil101	629	11570.4	1993.3	1890
rat195	2323	55772.0	4026.8	4024
rd400	15281	232955.0	143113.6	136158
ftv170	2755	5548.0	7146.0	7146
rbg323	1326	17136.0	5619.8	5591
rbg358	1163	23060.1	6279.8	5985
rbg403	2465	22937.2	7398.8	7295
rbg443	2720	30599.6	8060.4	7776

Table 2: Test Results

9 Discussion

To analyze and compare the reported results we will consider existing, manually derived heuristics for the TSP.

Firstly, Johnson et. al. [6] report on the results of Cycle Patching (P), Kanellakis-Papadimitriou (KP), Zhang's heuristic (Z) and Helsgaun's Heuristic (H) for the asymmetric TSP.

Secondly, Glover et. al. [7] report on the results of the Greedy algorithm (GR), Random Insertion (RI), Karp-Steele patching (KSP), a modified Karp±-Steele patching heuristic (GKS), a recursive path contraction algorithm (RPS), and a contract heuristic (COP) on both symmetric and asymetric problems.

Finally, Halim and Ismail [8] report on the results of the Nearest Neighbor algorithm (NN), a Genetic Algorithm (GA), Simulated Annealing (SA), Tabu Search (TS), Ant Colony Optimization (ACO), and Tree Physiology Optimization (TPO) on both symmetric and asymetric problems.

The performance - in terms of % over optimal distances - for each of the algorithms above is summarized in table 3. Where there are no reported results for an algorithm, the cell is left empty. In addition, the results for the Genetic Programming approach used in this assignment (GP) are summarized in the final column of the table.

Instance	% Over Optimal																
	Johnson et. al. [6]				Glover et. al. [7]						Halim and Ismail [8]						
	P	**KP**	**Z**	**H**	**GR**	**RI**	**KSP**	**GKS**	**RPS**	**COP**	**NN**	**GA**	**SA**	**TS**	**ACO**	**TPO**	**GP**
ch150	-	-	-	-	-	-	-	-	-	-	8.42	7.30	8.18	5.12	12.60	6.35	408.10
eil51	-	-	-	-	-	-	-	-	-	-	18.56	6.60	3.08	3.08	9.73	2.64	148.59
eil101	-	-	-	-	-	-	-	-	-	-	17.01	9.04	6.86	6.14	19.70	7.31	200.47
rat195	-	-	-	-	-	-	-	-	-	-	13.15	3.94	9.25	2.19	6.12	10.78	73.22
rd400	-	-	-	-	-	-	-	-	-	-	19.23	8.42	10.05	35.62	26.03	19.04	791.02
ftv170	1.38	4.44	0.36	0.00	32.05	28.97	2.40	1.38	25.66	3.59	-	-	-	-	-	-	159.38
rbg323	0.00	0.78	0.00	0.00	8.52	29.34	0.00	0.00	0.53	0.00	-	-	-	-	-	-	321.64
rbg358	0.00	1.50	0.00	0.00	7.74	42.48	0.00	0.00	2.32	0.26	-	-	-	-	-	-	414.61
rbg403	0.00	0.22	0.00	0.00	0.85	9.17	0.00	0.00	0.69	0.20	-	-	-	-	-	-	195.94
rbg443	0.00	0.11	0.00	0.00	0.92	10.48	0.00	0.00	0.00	0.00	-	-	-	-	-	-	185.88

Table 3: Comparison of Results

In comparing the results in table 3, it is obvious that the generation constructive hyper-heuristic explored in this assignment provides poor solutions in comparison to existing manually derived heuristics . However, considering that minimal experimentation was done with regards to parameter optimization and implementation variations, it is still worthwhile to further explore genetic programming in generating constructive low-level heuristics. Future research may also expand upon the execution times of these approaches, which were reported on in this assignment, but not discussed.

References

[1] N. Pillay and D. Beckedahl, "Evohyp-a java toolkit for evolutionary algorithm hyper-heuristics," in *2017 IEEE Congress on Evolutionary Computation (CEC)*, pp. 2706–2713, IEEE, 2017.

[2] G. Reinelt, "Tsplib—a traveling salesman problem library," *ORSA journal on computing*, vol. 3, no. 4, pp. 376–384, 1991.

[3] J. R. Koza and J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*, vol. 1. MIT press, 1992.

[4] B. L. Miller, D. E. Goldberg, *et al.*, "Genetic algorithms, tournament selection, and the effects of noise," *Complex systems*, vol. 9, no. 3, pp. 193–212, 1995.

[5] J. R. Koza and R. Poli, "Genetic programming," in *Search methodologies*, pp. 127–164, Springer, 2005.

[6] D. S. Johnson, G. Gutin, L. A. McGeoch, A. Yeo, W. Zhang, and A. Zverovitch, "Experimental analysis of heuristics for the atsp," in *The traveling salesman problem and its variations*, pp. 445–487, Springer, 2007.

[7] F. Glover, G. Gutin, A. Yeo, and A. Zverovich, "Construction heuristics for the asymmetric tsp," *European Journal of Operational Research*, vol. 129, no. 3, pp. 555–568, 2001.

[8] A. H. Halim and I. Ismail, "Combinatorial optimization: comparison of heuristic algorithms in travelling salesman problem," *Archives of Computational Methods in Engineering*, vol. 26, no. 2, pp. 367–380, 2019.