RAIS_{TTP} Revisited to Solve Relaxed Travel Tournament Problem

Elizabeth Montero*
Elizabeth.Montero@inf.utfsm.cl

María-Cristina Riff†
Maria-Cristina.Riff@inf.utfsm.cl

Universidad Técnica Federico Santa María
Avenida España 1680
Valparaíso, Chile

ABSTRACT

We are interested in methods and strategies that allow us to simplify the code of bio-inspired algorithms without altering their performance. In this paper, we study an artificial immune algorithm specially designed to solve Relaxed Traveling Tournament Problems which has been able to obtain new bounds for some instances of this problem. We use the EvoCa tuner to analyze the components of the algorithm in order to discard some parts of the code. The results show that the filtered algorithm is able to solve the instances as well as does the original algorithm, and with this code we have obtained new bounds for some instances of the problem.

Categories and Subject Descriptors

I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic methods

Keywords

Tuning; Immune Algorithms; Code Refining

1. INTRODUCTION

In order to obtain a metaheuristic with good performance, we have to make several design decisions. The design process is usually iterative, and at each step, the involved components must be evaluated. On the other hand, the final code of the metaheuristic can be extremely complex. Thus, analyzing and understanding its results becomes difficult, and this is often a very time consuming task. Unexperienced designers usually tend to include more and more components during the iterative design process of a metaheuristic, without evaluating the usefulness of previously incorporated ones. Some previus work has been proposed in this area [13] motivating the importance of a proper definition of the metaheuristic design problem. As in [13] we propose to have a refining step, in order to help the designer to obtain a simpler design with similar performance. The motivation of this work is to analyze in-depth $RAIS_{TTP}$ (Artificial Immune System for the Relaxed Traveling Tournament Problem) which is an immune algorithm that has many components and which has been able to obtain new bounds for the relaxed traveling tournament problem[15]. Our goal is to refine $RAIS_{TTP}$ without loss performance (e.g. in terms of the solutions quality). It is important to remark that our goal is not to find the best solution to the relaxed traveling tournament problem, but to focus on the way to distinguish the components that are strictly required from the ones that are unnecessary in $RAIS_{TTP}$. We briefly revise published works related to this subject in section 2. In section 3 we formally define the Refining Metaheuristic Design (RMD) problem as configuration problem. After a brief description of the Relaxed Traveling Tournament Problem in section 4, we present the principal components of the immune algorithm $RAIS_{TTP}$ and we show how we use the Evolutionary Calibrator (EvoCa) [17] tuner that can work with categorical and numerical parameters in this RMD in section 6. We present the results of a set of experiments in section 7, as well as a statistical comparison between the results obtained using different versions of $RAIS_{TTP}$. Finally, we present the conclusions and future work in section 8. The contributions of this paper are: an in-depth analysis of the data generated by EvoCa to make better decisions to accomplish the refining design problem, a new version of the immune algorithm that solves RTTP and new bounds for the RTTP.

2. RELATED WORK

Most of the motivation of this work arises from literature on automated parameter tuning and algorithm configuration [9]. In automated algorithm configuration, the design space is defined by an algorithm scheme that contains a number of instantiable components, along with a discrete set of concrete choices for each of these. Some ideas have been proposed to use tuning methods for supporting metaheuristic design processes [13, 2]. We can also relate these approaches to hyper-heuristic methods [3], whose goal is to find a good design with sufficient quality.

On the other hand artificial immune algorithms have been widely applied to optimization problems the last two decades. Those applications include numerical optimization [19], constrained optimization problems [6], constraint satisfaction problems [18], combinatorial optimization problems: like

*Supported by Postdoctoral Fondecyt Project no. 3130754
†Supported by Fondecyt Project no. 1151456

GECCO '15, July 11 - 15, 2015, Madrid, Spain

DOI: http://dx.doi.org/10.1145/2739480.2754753

traveling salesman [7] and job-shop scheduling [5]; multi-objective optimization [23] and dynamic optimization problems [22].

The traveling tournament problem is a real challenge to both complete and incomplete combinatorial optimization techniques. Best complete techniques are able to find the optimum for small instances of up to 10 teams [20]. Finding the optimum requires a long execution time. For bigger instances the most successful methods are those based on metaheuristics. A simulated annealing for TTP is proposed in [1] where reheats and a strategic oscillation strategy are used to vary the constraint weight parameter during the search. This method was very successful for solving TTP obtaining several of the best solutions so far. Tabu search [10], Ant colony optimization [21], immune algorithms [14] and hybrid methods [11] have also been proposed. In the field of hyper-heuristics, an ant based hyper-heuristic is introduced in [4] and a learning hyper-heuristic is given in [12]. Both obtaining good results. Best known optima for the Relaxed Traveling tournament are published on the Michael Trick website[1].

3. THE REFINING METAHEURISTIC DESIGN PROBLEM

We can identify two problems when designing metaheuristics: The on-the-fly metaheuristic design problem (OMD) and the post-design problem or refining metaheuristic design problem (RMD). The first one occurs during the design process and is about the decisions to make when building efficient metaheuristics. In this paper we are interested in the refining problem. The Refining Metaheuristic Design Problem (RMD) occurs during post-design of the metaheuristic when we are interested in simplifying its code without performance loss. RMD can be stated as follows: given M, a target algorithm or metaheuristic, a set of parameters for the algorithm and a set of input data. The RMD consists in finding a reduced code for the metaheuristic which gives, at least, the same performance with the same input data than M. Figure 1 illustrates both problems. For the OMD problem, when we are building `My Code` we evaluate the inclusion of new components to improve its performance. After this evaluation a current code `Ccode` is obtained, which can follow a new procedure of inclusion of new components. For the RMD problem when the code is finished, it follows an evaluation step in order to identify, when this is possible, an alternative code which uses a reduced number of its components but has at least the same performance. In this work we map RMD to a configuration problem, where new parameters are introduced to the algorithm in order to identify alternative designs for M. More formally,

DEFINITION 3.1. *Given a metaheuristic code M, an instance of the problem consists in a 6-tuple*

$$P = (M, S, \Theta, \Pi, \kappa_{max}, M'),$$

where S is the set of new binary parameters introduced in M that allows to either turn on or off some components. M' is the modified version of M that includes S. Θ is the configurations space for M'. Π is the set of input problem instances, $g(\theta, \Pi)$ is a function that computes the expected gain (e.g., the quality of the solutions) of running M' using instance $\pi \in \Pi$ when using configuration θ. κ_{max} is a time out after which all instances of M' will be terminated if they are still running.
Any configuration $\theta \in \Theta$ is a candidate configuration of P. The gain of a candidate configuration θ is given by:

$$G_P(\theta) = mean_{\pi \in \Pi}(g(\theta, \pi))$$

This definition considers the mean of gain induced by $g(\theta, \pi)$, but any other statistic could be used instead (e.g. median, variance). Given G^M, the gain of metaheuristic M using its best parameter configuration, an alternative code defined by the configuration θ^* has a value $G_P(\theta)$ such that:

$$G_P(\theta^*) \geq G^M$$

We formally define an alternative design of the final metaheuristic code M as:

DEFINITION 3.2. *Given M a metaheuristic with a performance G^M and $M_1, \ldots, M_l$ alternative algorithms with performances $G_{M_1}(\theta_1), \ldots, G_{M_l}(\theta_l)$ for a maximization problem. M_i belongs to the* **set of alternative designs** *S_d, if and only if, $G_{M_i}(\theta_i) \geq G^M$. We define M_i as the* **best alternative design** *such that*

$$G_{M_i}(\theta_i) \geq G_{M_j}(\theta_j), \forall M_i, M_j \in S_d, i \neq j$$

During the post-design we are looking for a possible alternative code that allows the metaheuristic M' to solve the problems as M does with at least the same performance, but using a reduced number of its initial components. Given its stochastic nature, it is noteworthy to mention that the refined algorithm could show better performance than the initial one. Definitions are also applicable to minimization problems.

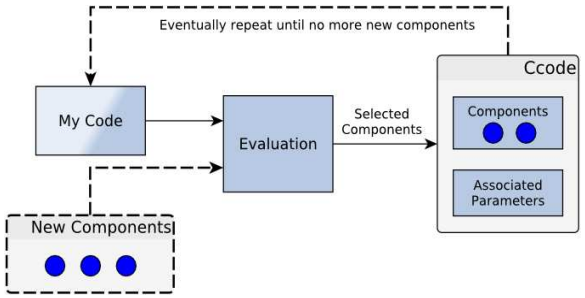

Figure 1: Illustration of the designing problems

4. RELAXED TRAVELING TOURNAMENT

The Traveling Tournament Problem [8] (TTP) is a timetabling problem which involves two issues: the assignment feasibility (tournament structure) and optimization (reducing distance traveled by all teams). The objective is, given a number of teams N and a distance matrix $D_{i,j}$, to find a double round robin tournament which minimizes the distance traveled. The *trip length* constraint establishes the minimum and the maximum number of consecutive home/away matches allowed during the tournament. Usually, the lower and upper boundaries are one and three respectively.
A solution must also satisfy the following constraints:

[1] http://mat.gsia.cmu.edu/TOURN/relaxed/

- Constraint 1 [No repeaters]: Matches that involve the same pair of teams (i, j) must not take place in consecutive rounds.

- Constraint 2 [Mirrored]: The first half of the tournament should be a single round robin tournament, the second is obtained by mirroring the first one.

The double round robin structure requires $2(N-1)$ rounds, $N/2$ matches must take place every round, each pair of teams must play twice during the tournament exchanging homes, and all teams must play only one match every round.

The goal in the TTP is to find a compact schedule: the number of time slots is equal to the number of games each team plays. This forces every team to play in every time slot. In the Relaxed Traveling Tournament problem, the schedule is not required to be compact and teams are allowed to have a fixed number K of byes, hence RTTP schedule length is $2(N-1)+K$. Byes are ignored in determining the length of a home stand or road trip, and in determining whether a repeater has occurred. The higher the value of K the more flexibility to find solutions with lower distance traveled.

5. $RAIS_{TTP}$ ALGORITHM

In this section we introduce $RAIS_{TTP}$ artificial immune algorithm to solve the Relaxed TTP. For more details refer to [15] were $RAIS_{TTP}$ was introduced.

5.1 Representation

Considering a set of teams $S_T = \{T_1, \ldots, T_N\}$ of N teams, and a set of rounds $S_R = \{R_1, \ldots, R_M\}$ of M rounds, solutions are represented using a matrix $TM_{N\text{X}M}$, where the absolute value of each element $|tm_{ij}|$ indicates the opponent of team T_i in the round R_j. More precisely,

- $|tm_{ij}| \in \{1, \ldots, N\}$, with $|tm_{ij}| \neq i$

- $tm_{ij} > 0$ when the match is at home

- $tm_{ij} < 0$ when the match is away.

$RAIS_{TTP}$ approach considers constraints related to the double round robin stucture as hard constraints, and the moves take their satisfaction into account. The non structural constraints of *No repeaters* and *trip length* are managed by a penalization on the fitness function.

5.2 Fitness Function

In order to guide the algorithm to find good quality solutions which also satisfy the constraints, the algorithm uses equation 1 which computes the penalty function W_c for the constraint c. In this equation $\bar{d}$ is the average distance among teams and δ_c is the penalty factor associated to the dissatisfaction of the constraint c.

$$W_c = \bar{d} * (N-1) * \delta_c, \forall c = 1, 2. \quad (1)$$

Finally, equation 2 defines the fitness function F_{ttp}, considering all rounds, ϕ_i is the total traveled distance of team i and v_c is the number of violated non-structural constraints in a scheduling candidate.

$$F_{ttp} = \sum_{i=1}^{N} \phi_i + \sum_{c=1}^{2} v_c * W_c \quad (2)$$

5.3 $RAIS_{TTP}$ structure

$RAIS_{TTP}$ structure is shown in algorithm 1. Initial solutions are obtained using *"polygon method"* [16] to create single round robin structures (graph 1-factorization), that are mirrored and randomly assigned homes to complete the double round robin tournaments. Solutions are evaluated and cloned. cl_{rate} is the clonation rate which represents the percentage of the repertoire to be selected. Then, a set of clones is generated using the affinity proportional equation

$$C = \sum_{l=1}^{n} round\left(\frac{cl_{factor} * n}{l}\right),$$

where cl_{factor} is the clonal factor, n the repertoire size and l the antibody ranking. A hypermutation procedure is then performed. Hypermutation process is described in detail in section 5.5. Then, a new repertoire is created of the union between $Cells$ and the hypermutated population P_c. The selection of cells does not only consider their fitness but it also considers their diversity. Diversity of each solution in P_c is measured by the average of the hamming distance between its home/away pattern and the home/away pattern of the solutions already members of the selected cells set. $r_{rate} * n$ new solutions are included each iteration.

Algorithm 1: $RAIS_{TTP}$

```
1  Cells ← InitializePopulation(n) ;
2  i ← 1 ;
3  repeat
4  |  CalculateFitness (Cells) ;
   |  /*Select best cl_rate * n antibodies from Cells*/
5  |  B_a ← SelectBestAntibodies(cl_rate * n, Cells) ;
   |  /*Generate C clones of each antibody in B_a */
6  |  P_c ← GenerateClones(C, B_a) ;
7  |  P_c ← Hypermutation(P_c);
8  |  P_c ← P_c ∪ Cells;
9  |  Cells ← DiversitySelection(P_c, n);
10 |  Cells ← Cells+ GenerateNew(r_rate * n);
11 until termination_condition;
12 return;
```

5.4 Moves

The algorithm uses seven moves. *Swap Homes*, *Swap Rounds*, *Swap Teams*, *Partial Swap Rounds* and *Partial Swap Teams* have been proposed for a Simulated Annealing algorithm in [1]. *SwapByesMatches* and *GroupingAwayByes* were introduced in [14]. *GroupingAwayByes* is oriented to the reduction of traveling costs. As the cost of a team schedule corresponds to the cost of its away matches (trips), the idea is to try to group its trips to reduce the traveling costs.

- `SwapTeams`(T_i,T_j): Exchanges teams T_i and T_j matches, except for the ones which involve themselves.

- `SwapHomes`(T_i,T_j): Exchanges teams T_i and T_j matches home (matrix elements sign).

- `SwapRounds`(R_i,R_j): Exchanges rounds R_i and R_j schedules.

- `PartialSwapTeams`(T_i,R_j,R_k): Exchanges teams T_i and T_j matches for a round R_k.

- `PartialSwapRounds`(T_i,R_j,R_k): Exchanges round R_j match for R_k round match for a team T_i, .
- `SwapByesMatches`(T_i,T_j): Exchanges teams T_i and T_j from their current round to a new one were thay have a day off.
- `GroupingAwayByes` (T_i): Tries to group the trips of the team T_i using the byes in order to reduce the traveling costs.

5.5 Hypermutation procedure

In $RAIS_{TTP}$, the antibody ranking is used as a hypermutation level indicator. It determines the number of moves to apply to each clone. Algorithm 2 shows the procedure. It uses an exhaustive selection process from the solutions obtained by applying each of the six moves to the same candidate solution. The selected solution follows the `GroupingAwayByes` in order to try to improve it.

Algorithm 2: `Hypermutation`(P_c)

```
forall the s ∈ Pc do
    /*Ranking to determine hypermutation level */
    for x ← 1 to ranking do
        for y ← 1 to 6 do
            s*_y ← op_y(s);
        end
        sol ← Best(s*_1, ..., s*_6);
        s*_7 ←GroupingAwayByes(sol);
        if Better(s*_7, sol) then
            sol ← s*_7;
        end
    end
    s ← sol;
end
return;
```

6. REFINING $RAIS_{TTP}$

The goal of our experiments is to analyze which components of the hypermutation process are required. Thus, we evaluate the seven moves used by the $RAIS_{TTP}$ hypermutation procedure.

6.1 Refining strategy

In order to apply a tuner to this evaluation process we added seven parameters to $RAIS_{TTP}$. We have associated one binary parameter to each move. Algorithm 3 shows the pseudocode of our refining on hypermutation process.

In our approach the value of parameter $UseST$ indicates either to use or not use `SwapTeams`. The same is interpreted for parameters $UseSH$, $UseSR$, $UsePST$ $UsePSR$ $UseSBM$ and $UseGAB$ related to the remaining moves analyzed. For all these parameters a value 0 means that the associated move can be deleted from the code without loss performance.

From algorithm 3, the use of each move requires the evaluation of a new solution, hence, the more moves the $RAIS_{TTP}$ uses, the more computational effort is required for each iteration. In our approach we use EvoCa tuner in order to determine simpler codes than $RAIS_{TTP}$ requiring the same computational effort but having similar performance.

Algorithm 3: `Refining Hypermutation`(P_c)

```
forall the s ∈ Pc do
    /*Ranking to determine hypermutation level */
    for x ← 1 to ranking do
        for y ← 1 to 6 do
            /*Value of each binary parameter
              determines the use of related move */
            if UseOp_y then
                s*_y ← op_y(s);
            end
        end
        sol ← Best(s*_1, ..., s*_6);
        /*Value of parameter UseGAB determines
          the use of GroupingAwayByes          */
        if UseGAB then
            s*_7 ←GroupingAwayByes(sol);
        end
        if Better(s*_7, sol) then
            sol ← s*_7;
        end
    end
    s ← sol;
end
return;
```

6.2 Evolutionary Calibrator

EvoCa is an evolutionary algorithm that works with a population of parameter calibrations. It uses a wheel-crossover that constructs one child calibration from the whole population. The child replaces the worst calibration in the current population. It also uses a hill climbing procedure for mutation. The child generated by mutation replaces the second worst calibration in the current population, when a better parameter calibration is found. For a detailed description of EvoCa refer to [17].

7. EXPERIMENTS

In our experiments we use $RAIS_{TTP}$'s authors implementation. For these experiments we have considered the set of *National League* (NL) traveling tournament problem instances: $NL4$ to $NL16$ ranging from 4 to 16 teams. Moreover, we have considered experiments with $K = 1$, $K = 2$ and $K = 3$. We used EvoCa's authors implementation with a population of 10. Sources for our experiments are available in our website[2].

7.1 Tuning process

The tuning process considered a total of 10 parameter values to tune. Here, we tune 3 original $RAIS_{TTP}$ parameters defined by its authors: cl_{rate} represents the percentage of solutions to be selected for clonation, cl_{factor} is the clonal factor related to the amount of clones that will be generated from each solution and r_{rate} represents the percentage of new solutions included each iteration. The seven binary parameters previously defined were also tuned in this process. 20 independent tuning processes were executed in order to deal with stochastic nature of tuning method. Each tuning process considered the entire NL set of instances. A maximum budget of $10K$ $RAIS_{TTP}$ executions was fixed for each

[2]comet.informaticae.org

tuning process. Each $RAIS_{TTP}$ considered a termination criterion of $7K$ evaluations.

7.2 Results

Table 1 shows the results obtained in 20 different tuning processes. In each tuning process 10 parameter values were tuned. Seven of these parameter values can be considered design decisions and the remaining three are numerical parameter of $RAIS_{TTP}$. Hence, each tuning process can be seen as a competition between different design options (each one using proper $RAIS_{TTP}$ parameter values) and its output represents the best design decision with the best parameter values. It is important to point out that design decisions and tuning of $RAIS_{TTP}$ parameters can not be made independently because the determination of the final performance of $RAIS_{TTP}$ depends on the way they interact.

To calculate the gain $G_P(\theta)$ of these algorithms they were executed 100 times in each instance. For these experiments, the number of evaluations was fixed to $7K$ for $NL4$ and $NL6$, $210K$ for $NL8$, $NL10$, $NL12$ and $NL14$; and $1400K$ for $NL16$. Relative distance to best performance was calculated for each instance. Column $G_P(\theta)$ in table 1 shows the average of relative distance to best performance considering the entire set of instances. For a minimization problem the algorithm with lower gain value shows the best performance.

From table 1 we can observe that in 25% of cases the best design is not use `SwapRounds` and `SwapByesMatches` operators. The diversity of the results in the performance for this design is due to the stochastic nature of Evoca, because in different runs it identifies different parameter's values. However, each run uses different seeds and it follows an internal competition process among the set of the alternative designs. Thus, the selected design is the best option after the complete evaluation process of each run. We can also observe that the original design has been selected as the best of the run only once ($E1$). From these results we identify the design selected more times and from those designs $E12$ has the best parameter's values.

In order to obtain the best performance for the selected design a complete new tuning process could be done to obtain the three best numerical parameter values of the new design of $RAIS_{TTP}$. The tuning method used by the authors of $RAIS_{TTP}$ was an exhaustive one, where all possible values have been tested.

Table 3 compares the performance of algorithm $RAIS_{TTP}$ ($E12$) with the original $RAIS_{TTP}$ for each instance studied. For these experiments we executed $RAIS_{TTP}$ using values tuned by authors in [15] and shown in table 2. Table 3 shows the average and best performance of 100 independent executions of the original $RAIS_{TTP}$ versus $RAIS_{TTP}$ ($E12$). Last column shows the p-value of Wilcoxon signed rank test comparing the performance of both algorithms for each instance. Considering a confidence level of 95% statistically different performance is presented in bold face.

Table 2: Parameter values used in [15]

Instance	cl_{rate}	cl_{factor}	r_{rate}
All	1.0	1.0	0.3

From these results we can conclude that design $E12$ allows to simplify $RAIS_{TTP}$ algorithm finding no statistically significant difference in 12 cases and a statistical better performance in 9 cases.

Graphs in figures 2-7 show box-plots comparing the results of $RAIS_{TTP}$ and $RAIS_{TTP}(E12)$ considering $K = 1$, $K = 2$ and $K = 3$ for each instance. We did not include a graph for $NL4$ because both $RAIS_{TTP}$ versions found always the same performance in this instance.

Here we can observe that for instances $NL6$-$NL10$ the performance of both versions of $RAIS_{TTP}$ are very similar, but in larger instances, $RAIS_{TTP}$ ($E12$) clearly outperforms the original design considering all values of K.

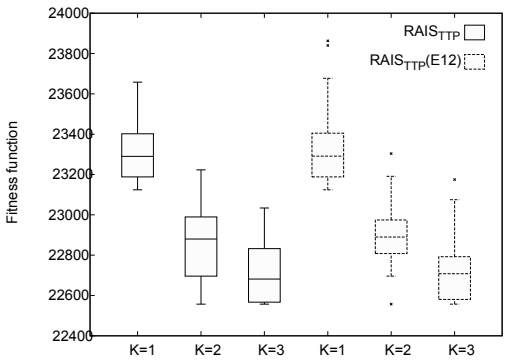

Figure 2: $RAIS_{TTP}$ **versus** $RAIS_{TTP}(E12)$ **for** $NL6$

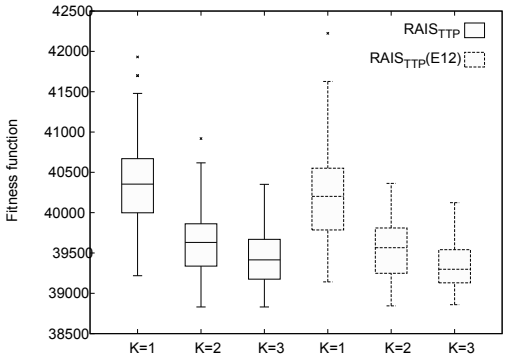

Figure 3: $RAIS_{TTP}$ **versus** $RAIS_{TTP}(E12)$ **for** $NL8$

Table 4 compares the performance of $RAIS_{TTP}$ ($E12$) with the best bounds published. Here, we can observe that the new design for $RAIS_{TTP}$ was able to find new bounds for $NL10$ with $K = 1$ and $K = 3$.

Graphs in figures 8 and 9 show convergence of $RAIS_{TTP}$ and $RAIS_{TTP}(E12)$ when they were able to find their best solutions in table 3. Logarithmic scale was used for x axis in order to improve the reading.

Figure 8 shows the best behaviour of each algorithm to solve $NL10$ with $K = 1$. Because of each algorithm has shown its best behaviour using different seeds, the initial populations are different. We observe that $RAIS_TTP(E12)$ despite starting with poorer population is able to converge to a better solution than $RAIS_TTP(E12)$. $RAIS_TTP(E12)$ can perform a higher number of evaluations given its simplicity compared to the original algorithm.

Table 1: Algorithms and their performances

Id.	cl_rate	cl_factor	r_{rate}	$useST$	$useSH$	$useSR$	$usePST$	$usePSR$	$useSBM$	$useGAB$	$G_P(\theta)$
E0	0.90	0.80	0.32	1	0	0	1	1	0	1	1.96
E1	0.82	0.30	0.00	1	1	1	1	1	1	1	2.09
E2	1.00	0.20	0.30	1	1	1	1	1	0	1	2.19
E3	0.89	0.18	0.53	1	1	0	1	1	0	1	2.49
E4	0.60	1.00	0.40	1	1	0	1	1	1	1	1.09
E5	0.80	1.00	0.28	1	1	0	1	1	1	0	2.13
E6	1.00	1.00	0.20	1	1	0	0	1	0	1	1.19
E7	0.50	0.90	0.60	1	1	0	1	1	0	1	1.80
E8	0.40	0.90	0.05	1	1	0	1	1	0	1	0.85
E9	0.50	1.00	0.20	1	0	0	1	1	1	1	1.83
E10	0.84	0.90	0.50	1	1	1	1	1	0	0	2.37
E11	0.51	0.97	0.00	1	1	0	0	1	0	1	1.01
E12	**0.78**	**1.00**	**0.20**	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0.62**
E13	0.80	0.50	0.00	0	1	0	1	1	1	0	5.34
E14	0.84	0.65	0.56	1	1	0	1	1	0	1	1.77
E15	0.91	0.19	0.20	1	0	1	1	1	1	1	2.97
E16	0.80	0.40	0.50	1	1	0	1	1	1	0	3.52
E17	0.40	0.85	0.00	1	0	1	0	1	1	1	2.96
E18	1.00	0.90	0.00	1	1	1	1	1	0	1	0.87
E19	0.70	0.30	0.60	0	1	1	1	1	0	0	5.60
% usage				90	80	35	85	100	40	75	

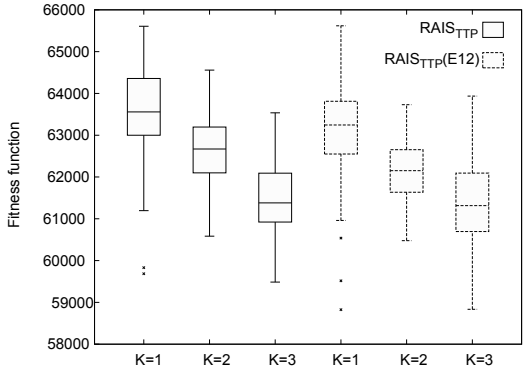

Figure 4: $RAIS_{TTP}$ **versus** $RAIS_{TTP}(E12)$ **for** $NL10$

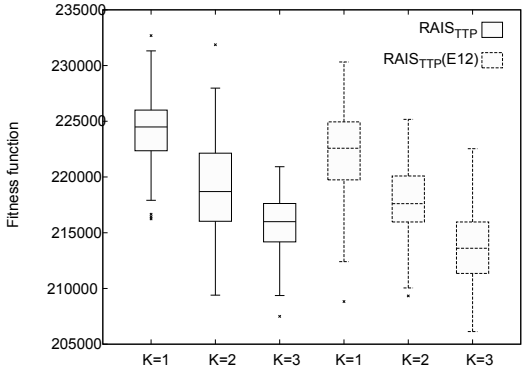

Figure 6: $RAIS_{TTP}$ **versus** $RAIS_{TTP}(E12)$ **for** $NL14$

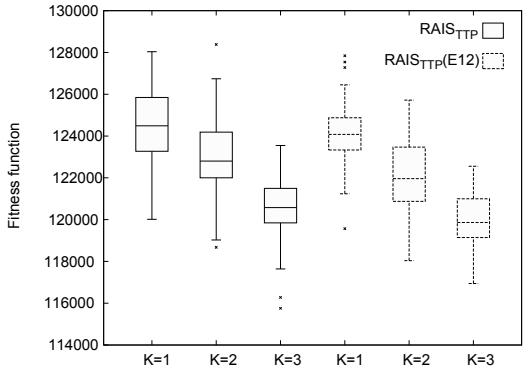

Figure 5: $RAIS_{TTP}$ **versus** $RAIS_{TTP}(E12)$ **for** $NL12$

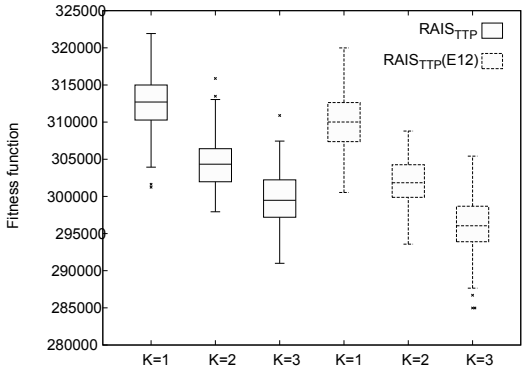

Figure 7: $RAIS_{TTP}$ **versus** $RAIS_{TTP}(E12)$ **for** $NL16$

Table 3: Performance comparison: $RAIS_{TTP}$ versus $RAIS_{TTP}(E12)$

Instance		$RAIS_{TTP}$		$RAIS_{TTP}(E12)$		p-value
		Average	Best	Average	Best	
NL4	K=1	8160.0	8160	8160.0	8160	-
	K=2	8160.0	8160	8160.0	8160	-
	K=3	8044.0	8044	8044.0	8044	-
NL6	K=1	23308.7	23124	23322.9	23124	0.75
	K=2	22861.9	22557	22895.9	22557	0.08
	K=3	22700.5	22557	22705.7	22557	0.89
NL8	K=1	40343.7	39218	40213.9	39142	0.09
	K=2	39625.1	38831	39569.9	38845	0.29
	K=3	39432.0	38831	39355.7	38859	0.08
NL10	K=1	63542.8	59686	**63160.8**	58825	**0.01**
	K=2	62653.7	60584	**62095.6**	60477	**0.00**
	K=3	61526.9	59486	61407.5	58834	0.48
NL12	K=1	124457.4	120016	124107.1	119570	0.08
	K=2	123027.6	118674	**122089.0**	118037	**0.00**
	K=3	120564.7	115755	**119955.7**	116942	**0.00**
NL14	K=1	224272.7	216236	**222197.0**	208823	**0.00**
	K=2	219010.7	209402	218020.0	209331	0.09
	K=3	215872.1	207495	**213782.1**	206134	**0.00**
NL16	K=1	312448.2	301232	**309895.2**	300531	**0.00**
	K=2	304606.7	297936	**301840.7**	293575	**0.00**
	K=3	299642.1	290998	**296065.9**	284982	**0.00**

Table 4: Performance of calibration $E12$

Instance		Best known	E12	
			Average	Best
NL4	K=1	8160	8160.0	8160
	K=2	8160	8160.0	8160
	K=3	8044	8044.0	8044
NL6	K=1	22642	23322.9	23124
	K=2	22557	22895.9	22557
	K=3	22557	22705.7	22557
NL8	K=1	39128	40213.9	39142
	K=2	38761	39569.9	38845
	K=3	38670	39355.7	38859
NL10	K=1	59425	63160.8	**58825**
	K=2	59373	62095.6	60477
	K=3	59436	61407.5	**58834**
NL12	K=1	108629	124107.1	119570
	K=2	108629	122089.0	118037
	K=3	108629	119955.7	116942
NL14	K=1	183354	222197.0	208823
	K=2	183354	218020.0	209331
	K=3	183354	213782.1	206134
NL16	K=1	249477	309895.2	300531
	K=2	249477	301840.7	293575
	K=3	249477	296065.9	284982

Figure 9 shows the convergence process for instance $NL10$ with $K = 3$. The same effect as in figure 8 can be noticed here. The use of moves `SwapRounds` and `SwapByesMatches` leads to a stagnation of the search , while $RAIS_{TTP}(E12)$ shows a similar behavior in first steps, but it is still able to obtain important improvements in the last steps of the search.

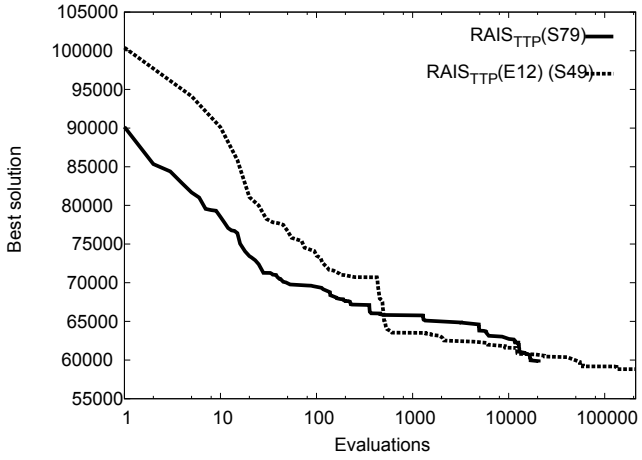

Figure 8: Convergence of $RAIS_{TTP}$ versus $RAIS_{TTP}(E12)$ for $NL10$ with $K = 1$

8. CONCLUSIONS AND FUTURE WORK

In this work we did an in-depth analysis of the immune algorithm $RAIS_{TTP}$, that was specially designed to solve instances of the relaxed travel tournament problem. We addressed the refining design problem, that is a post-design problem. We are interested in simplifying the $RAIS_{TTP}$ code without performance loss. From this study we have defined a simpler code ($RAIS_{TTP}$ $(E12)$) that shows no statistical significance difference in performance. Moreover, using this refined code we have obtained new bounds for the RTTP. The fact we have obtained a simpler algorithm does not mean that the original metaheuristic is not a good code. The classical incremental process of designing metaheuristics

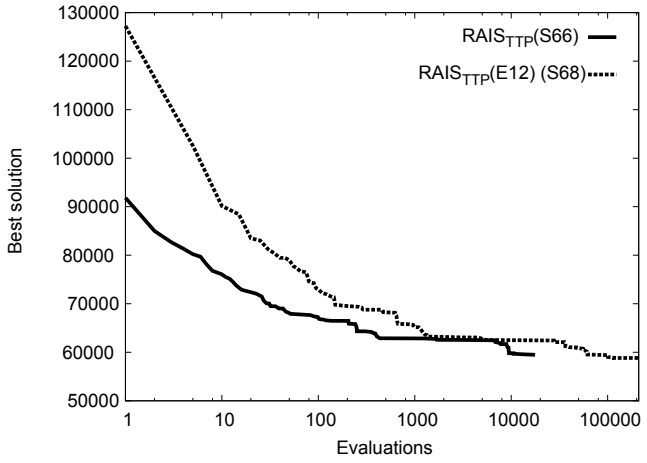

Figure 9: Convergence of $RAIS_{TTP}$ versus $RAIS_{TTP}(E12)$ for $NL10$ with $K = 3$

can produce a complex code, in which the interaction among many components is unknown. The key idea when searching for a refined code is to better understand the code behavior as well as the problem.

As future work we want to better analyze $RAIS_{TTP}$ using the refinement method to study new components in order to improve its performance in more complex instances.

Acknowledgment

We thank Mrs. Leslie Pérez-Cáceres for her support with $RAIS_{TTP}$ implementation.

9. REFERENCES

[1] A. Anagnostopoulos, L. Michel, P. V. Hentenryck, and Y. Vergados. A Simulated Annealing Approach to the Traveling Tournament Problem. *Journal of Scheduling*, 9(2):177–193, 2006.

[2] L. C. T. Bezerra, M. López-Ibáñez, and T. Stützle. Automatic design of evolutionary algorithms for multi-objective combinatorial optimization. In *PPSN*, volume 8672 of *LNCS*, pages 508–517. 2014.

[3] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In *Handbook of Metaheuristics*, volume 57, pages 457–474. 2003.

[4] P.-C. Chen, G. Kendall, and G. Berghe. An ant based hyper-heuristic for the travelling tournament problem. In *IEEE Symposium on Computational Intelligence in Scheduling*, pages 19–26, 2007.

[5] C. A. Coello-Coello, D. Cortés-Rivera, and N. Cruz-Cortés. Use of an artificial immune system for job shop scheduling. In *Artificial Immune Systems*, volume 2787 of *LNCS*, pages 1–10. 2003.

[6] N. Cruz-Cortés, D. Trejo-Pérez, and C. A. Coello-Coello. Handling constraints in global optimization using an artificial immune system. In *Artificial Immune Systems*, volume 3627 of *LNCS*, pages 234–247. 2005.

[7] L. N. de Castro and F. J. V. Zuben. Learning and optimization using the clonal selection principle. *IEEE Transactions on Evolutionary Computation*, 6(3):239–251, 2002.

[8] K. Easton, G. Nemhauser, and M. Trick. The traveling tournament problem description and benchmarks. In *Principles and Practice of Constraint Programming*, volume 2239 of *LNCS*, pages 580–585, 2001.

[9] A. E. Eiben and S. K. Smit. Parameter Tuning for Configuring and Analyzing Evolutionary Algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.

[10] L. D. Gaspero and A. Schaerf. A composite-neighborhood tabu search approach to the traveling tournament problem. *Journal of Heuristics*, 13(2):189–207, 2007.

[11] A. Lim, B. Rodrigues, and X. Zhang. A simulated annealing and hill-climbing algorithm for the traveling tournament problem. *European Journal of Operational Research*, 174(3):1459–1478, 2006.

[12] K. V. M. Misir, T. Wauters and G. V. Berghe. A new learning hyper-heuristic for the traveling tournament problem. In *MIC 2009: The VIII Metaheuristics International Conference*, 2009.

[13] E. Montero and M.-C. Riff. Towards a method for automatic algorithm configuration: A design evaluation using tuners. In *PPSN*, volume 8672 of *LNCS*, pages 90–99. 2014.

[14] L. Pérez and M.-C. Riff. New bounds for the relaxed traveling tournament problems using an artificial immune algorithm. In *CEC*, pages 873–879, 2011.

[15] L. Pérez-Cáceres and M.-C. Riff. Solving scheduling tournament problems using a new version of clonalg. *Connection Science*, 27(1):5–21, 2015.

[16] C. C. Ribeiro and S. Urrutia. Heuristics for the mirrored traveling tournament problem. *European Journal of Operational Research*, 179(3):775–787, 2007.

[17] M.-C. Riff and E. Montero. A new algorithm for reducing metaheuristic design effort. In *CEC*, pages 3283–3290, Cancún, México, June 2013.

[18] M.-C. Riff, M. Zúñiga, and E. Montero. A graph-based immune-inspired constraint satisfaction search. *Neural Computing and Applications*, 19(8):1133–1142, 2010.

[19] J. Timmis, C. Edmonds, and J. Kelsey. Assessing the performance of two immune inspired algorithms and a hybrid genetic algorithm for function optimisation. In *CEC*, volume 1, pages 1044–1051, 2004.

[20] D. C. Uthus, P. J. Riddle, and H. Guesgen. Solving the traveling tournament problem with iterative-deepening a*. *Journal of Scheduling*, 15(5):601–614, 2012.

[21] D. C. Uthus, P. J. Riddle, and H. W. Guesgen. An ant colony optimization approach to the traveling tournament problem. In *GECCO*, pages 81–88. ACM, 2009.

[22] J. Walker and S. Garrett. Dynamic function optimisation: comparing the performance of clonalg and evolution strategies. In *Artificial Immune Systems*, volume 2787 of *LNCS*, pages 273–285, 2003.

[23] Z. Zhang. Immune optimization algorithm for constrained nonlinear multiobjective optimization problems. *Applied Soft Computing*, 7(3):840–857, 2007.