An Adaptive and Near Parameter-Free BRKGA Using Q-Learning Method

Antonio Augusto Chaves and Luiz Henrique Nogueira Lorena

Univ Fed of São Paulo, São José dos Campos, Brazil
antonio.chaves@unifesp.br
http://www.ict.unifesp.br/antoniochaves/

Abstract. The Biased Random-Key Genetic Algorithm (BRKGA) is an efficient metaheuristic to solve combinatorial optimization problems but requires parameter tuning so the intensification and diversification of the algorithm work in a balanced way. There is, however, not only one optimal parameter configuration, and the best configuration may differ according to the stages of the evolutionary process. Hence, in this research paper, a BRKGA with Q-Learning algorithm (BRKGA-QL) is proposed. The aim is to control the algorithm parameters during the evolutionary process using Reinforcement Learning, indicating the best configuration at each stage. In the experiments, BRKGA-QL was applied to the symmetric Traveling Salesman Problem, and the results show the efficiency and competitiveness of the proposed algorithm.

Keywords: Genetic Algorithm · Q-Learning · Reinforcement Learning · Parameter control.

1 Introduction

Metaheuristics are important to find good solutions for NP-hard combinatorial optimization problems. There are a large number of metaheuristics in the literature with different search strategies. Some decisions, however, must be made by the developers [21]:

- Solution representation;
- Objective function;
- Constraints (feasible and infeasible solutions);
- Neighborhood structures;
- Initial solution;
- Parameter tuning.

These decisions have many alternatives, and appropriate choices are essential so the metaheuristic can find good solutions to a specific optimization problem. Another natural question that arises is: which metaheuristic to use given a problem? According to [23], it depends on the responses given to the questions previously presented since no algorithm will outperform all others on all problems and instances (*no free lunch theorem*).

Metaheuristic parameters have a direct influence on its efficiency and effectiveness. Consequently, their values should be tuned before the metaheuristic is used. Sevaux et al. [18] define the configuration of a metaheuristic as the specification of values for each parameter and the control flow (order of the different metaheuristic components). Finding optimal parameter values in a reasonable time is a difficult process that becomes even harder if other features, like control flow, are considered.

Chaves et al. [2] designed a metaheuristic with online parameter control that requires less configuration effort to tuning the method, called Adaptive Biased Random-Key Genetic Algorithm (A-BRKGA). This technique is a variation of the Biased Random-Key Genetic Algorithm (BRKGA) [5], a metaheuristic composed of problem-dependent and independent components. Examples of problem-dependent components are the objective and the local search procedures, while the remaining of the technique's components are problem independent and can be reused.

Although A-BRKGA has proven efficient, several decisions were defined a priori and maintained during the search process. These decisions refer to deterministic rules for adapting parameters, crossover and mutation operators, and the order that local search heuristics are applied. Thus, this work proposes a new BRKGA variant, called BRKGA-QL, that uses concepts of Reinforcement Learning [20] to create an algorithm with improved robustness and easier to configure. The Q-Learning algorithm [22] was used to control the parameters of the BRKGA during the evolutionary process.

In the literature, the Q-Learning algorithm was used with success to control the parameters of some metaheuristics like Variable Neighborhood Search [10, 13], Differential Evolution [6], and Simulated Annealing [17]. In the context of the current work, this online approach to optimize control parameters and flow may minimize the effects of bad decisions made during the tuning process. The use of BRKGA is justified because its problem independent components reduce development effort, and this technique achieved success in solving different optimization problems in the literature.

The remainder of the research article is organized as follows. Section 2 presents a brief description of the BRKGA and Q-Learning methods. In Section 3, the BRKGA-QL applied to solve the Traveling Salesman Problem (TSP) is described. Section 4 reports the computational experiments and Section 5 makes concluding remarks.

2 Methods

The BRKGA is an evolutionary algorithm that provides a general and reusable framework to solve different combinatorial optimization problems, and the Q-Learning (QL) is considered one of the widely used algorithms in Reinforcement Learning. The BRKGA method is detailed in subsection 2.1 and the Q-Learning is presented in subsection 2.2.

2.1 BRKGA

BRKGA was proposed by [5] as a heuristic method for combinatorial optimization. It is a variation of the random-key genetic algorithm [1] that always uses one parent from the elite set for mating and bias the crossover operation in favor of the elite parent.

The pseudocode of BRKGA is presented in Algorithm 1. It works like any other genetic algorithm, where the aim is to improve a population of chromosomes (solutions) at every generation by applying genetic operators. The algorithm receives as parameters the population size (p), percentage of elites (p_e), percentage of mutants (p_m), probability of inheriting a key from the elite parent (ρ_e), and the total number of generations that the algorithm should execute (g_{total}).

Algorithm 1 Biased Random-Key Genetic Algorithm

```
1: procedure BRKGA(p, p_e, p_m, ρ_e, g_total)
2:   Generate Pop with p vectors of random-keys
3:   Evaluate and sort Pop by fitness
4:   Store the best solution sol*
5:   for g_total generations repeat
6:     Create Pop+ elite set (P_e) using p_e as guide
7:     Create Pop+ offspring set (P_o) using ρ_e as guide
8:     Create Pop+ mutant set (P_m) using p_m as guide
9:     Pop ← Pop+
10:    Evaluate and sort Pop by fitness
11:    Store the best solution sol*
12:  return sol*
13: end procedure
```

In lines 2-4 of the algorithm, the initial population is created, evaluated, and the chromosome with the best fitness is stored. Lines 5-11 correspond to the main loop of the algorithm that will execute for g_{total} generations. Inside this loop, the next-generation population (Pop^+) is created. Figure 1 illustrates the transition between populations that occurs at every generation of BRKGA.

In lines 6-8, the genetic operators are applied over Pop to generate Pop^+. At line 6, Pop is partitioned into elite set P_e (composed by $p * p_e$ chromosomes) and a non-elite set. P_e is copied directly to Pop^+. At line 7, an offspring set (P_o) is created using the *parametrized uniform crossover* operator [19] with ρ_e as a parameter, and copied to the next generation. Figure 2 shows an example of how the *parametrized uniform crossover* is used as mating operator in BRKGA. There are two parents in this type of crossover, one randomly selected from an elite and another from a non-elite set. A random number is generated for each random-key, and if this number is less than ρ_e the offspring obtains the key from the elite parent. Otherwise, it gets the key from the non-elite. Finally, at line 8, a set of $p * p_m$ mutants (P_m) is created and copied to the next generation.

At line 9, the current population (Pop) is updated with the new created Pop^+. The population chromosomes are evaluated by the decoder and Pop is sorted by fitness. Line 11 stores the best chromosome in the population. At line 12, the algorithm returns the best chromosome.

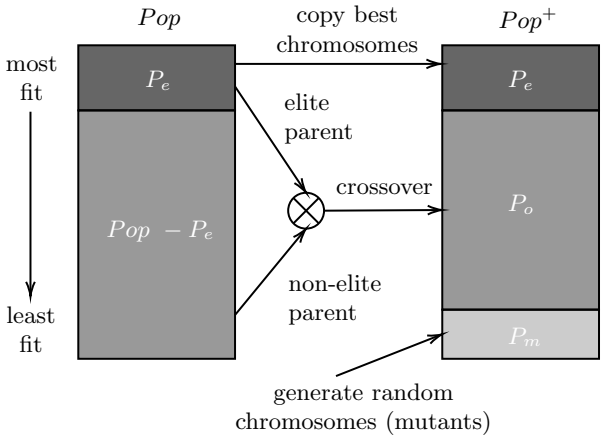

Fig. 1. Evolutionary process schema of the BRKGA (adapted from [5]).

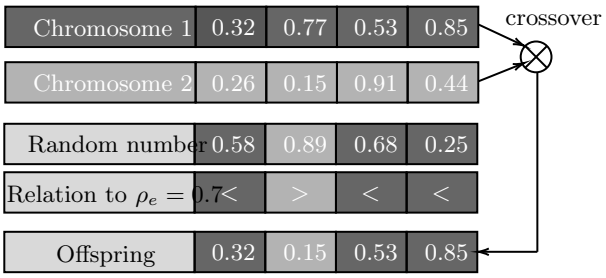

Fig. 2. Parametrized uniform crossover (adapted from [5]).

2.2 Q-Learning

Reinforcement Learning (RL) is a computational approach based on learning from interaction, using the maximization of a numerical reward signal as a goal. There are a few typical components in a RL scenario (Figure 3): the *agent*, *environment*, *reward* and *state*. In the typical RL optimization loop, the learning agent executes an action in the environment, and the environment returns as feedback a reward for executing such action and the new state of the agent [20].

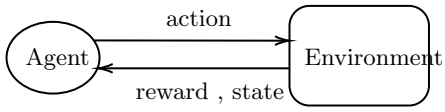

Fig. 3. RL components.

Q-Learning (QL) [22] is an RL algorithm that uses a finite Markov decision process framework to model the interaction between a learning agent and its environment in terms of states, actions, and rewards. It is a temporal-difference, incremental and model-free algorithm that works by successively improving the evaluation of the quality of the actions giving a particular state. Such evaluations are stored in tabular form (Q-Table):

$$
\begin{array}{rc}
 & \textit{Actions} \\
 & \begin{array}{cccc} a_1 & a_2 & \cdots & a_m \end{array} \\
\textit{States} & \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \\
\begin{array}{r} s_1 \\ s_2 \\ \vdots \\ s_n \end{array} &
\end{array}
$$

where $Q(s, a)$ corresponds to the value of taking action a when you are in state s. By incrementally updating the values of the Q-Table during the learning process, the algorithm defines a *policy* (mapping between actions to be taken given states). This *policy* defines how the agent behaves at a given time, and the main objective of the RL algorithm is to learn a policy that maximizes the cumulative reward it receives in the long run.

Algorithm 2 presents the pseudocode of the Q-Learning algorithm used in this work. The ϵ-greedy strategy was used to balance the exploration-exploitation of the method. The algorithm receives as input ϵ, the learning factor lf (that controls how fast the algorithm learns), and the discount factor df (used to balance immediate and future reward). The Q-Table is initialized at Line 2 (e.g. randomly). The external loop in Line 3 controls the number of episodes that the agent will interact with the environment. Inside the episode loop, the initial state of the agent (s) is initialized (Line 4) and the step loop (Line 5) begins. Inside the step loop, the agent should select an action based on the ϵ-greedy strategy (Line 6), this action will make the environment provide, at Line 7, a reward (r) and the agent's new state (s'). Line 8 is used to update the Q-Table using Bellman's equation for state-value function (Equation 1), and at Line 9 the current state of the agent is updated with s'. The stop condition of the step loop is when s is a terminal state.

$$
Q(s, a) \leftarrow Q(s, a) + lf \left[r + df * \max_{a'} Q(s', a') - Q(s, a) \right] \tag{1}
$$

The next section explains how the Q-Learning algorithm with ϵ-greedy strategy (Algorithm 2) was modified to work with BRKGA.

Algorithm 2 Q-Learning with ϵ-greedy strategy

```
1: procedure Q-Learning(ϵ, lf, df)
2:   Initialize Q-Table values
3:   for every episode repeat
4:     Choose a initial state s
5:     for each step of episode repeat
6:       Choose action a from A(s) using ϵ-greedy
7:       Take action a, observe the reward r and new state s'
8:       Update Q(s,a) using Equation 1
9:       s ← s'
10: end procedure
```

3 BRKGA-QL

In this section, the proposed algorithm is applied to solve the Traveling Salesman Problem (TSP). To solve other optimization problems users only need to develop the decoder and local search components. The Q-Learning algorithm will control, during the evolutionary process, both the BRKGA and Q-Learning parameters. From BRKGA, it will control the population size (p), the percentage of elites (p_e) and mutants (p_m), as well as the probability of inheriting a key from an elite parent (ρ_e). From Q-Learning, it will control the parameters: ϵ, lf (learning factor), and df (discount factor).

In the context of Q-Learning, the problem is formulated so there is only one state for each parameter. The action list of each parameter is given in Table I, consisting of recommended values from the literature for each BRKGA parameter [12], except population size. This value is defined by the Fibonacci sequence, a good candidate as it appears frequently throughout nature like in the population growth of certain animals [3].

Table 1. Action list of the parameters

Parameter	Action List
p	{233, 377, 610, 987, 1597, 2584}
p_e	{0.30, 0.25, 0.20, 0.15, 0.10}
p_m	{0.25, 0.20, 0.15, 0.10, 0.05}
ρ_e	{0.80, 0.75, 0.70, 0.65, 0.60, 0.55}
ϵ	{0.10, 0.15, 0.20, 0.25, 0.30}
lf	{0.2, 0.4, 0.6, 0.8, 1.0}
df	{0.2, 0.4, 0.6, 0.8, 1.0}

Algorithm 3 presents the pseudocode and Figure 4 shows the flowchart of the BRKGA-QL. First, the Q-Table is initialized with zeros (Line 2). The initial population of the BRKGA-QL is generated with p random-key vectors (chro-

mosomes or solutions) (Line 3). Each chromosome has $n + 1$ genes randomly generated with uniform probability between the interval $[0, 1]$. The value of n is defined by the number of cities in the TSP.

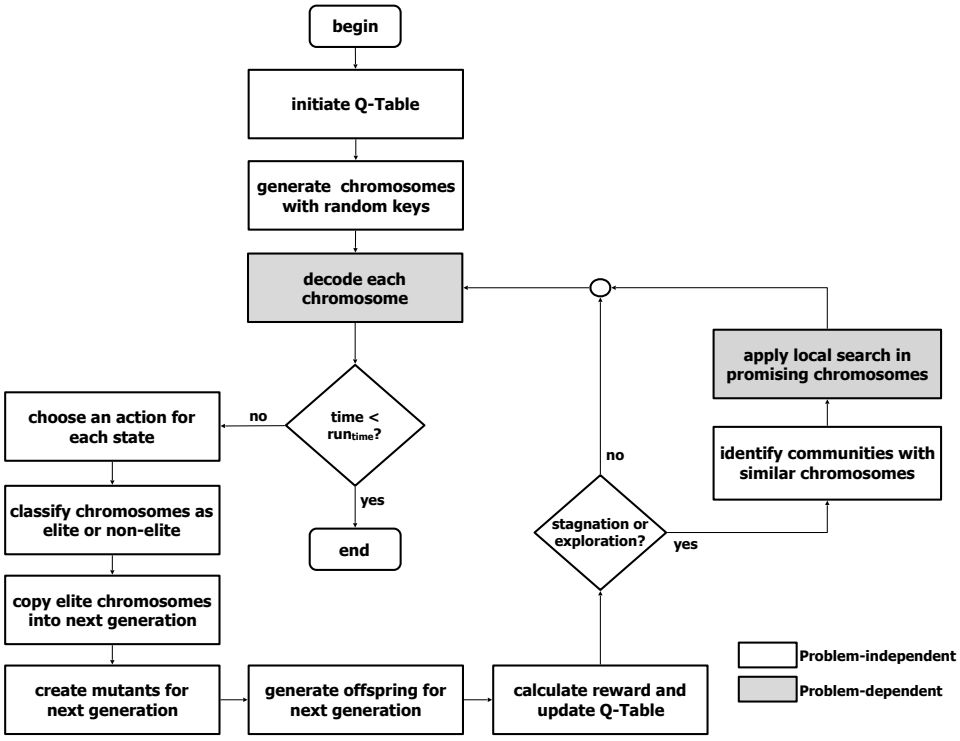

Fig. 4. BRKGA-QL Flowchart.

The decoder evaluates the fitness of each chromosome (Line 4) and stores the best solution (Line 5). In BRKGA-QL, users can implement k different decoders. A self-adapt schema to control the flow of the BRKGA was created by providing a new random-key at position $n + 1$, that is used to define which decoder the chromosome should use. Suppose a chromosome c, the id of the decoder is defined as $id = \lceil c[n + 1] * k \rceil$. This formula converts the random-key value to an id in the range $[1, \ldots, k]$. Thus, BRKGA-QL can explore the idea that different neighborhoods and similar random-key vectors can have different fitness values.

Each generation of the BRKGA-QL consists of lines 6 to 18. In Line 7, an action (value) from Q-Table is selected for each state (parameter) using the ϵ-greedy policy. The objective is to enable a trade-off between visiting more often good actions (intensification) and explore eventually actions that can lead to the discovery of new policies even better than those already existing (diversification).

Algorithm 3 BRKGA and Q-Learning

```
 1: procedure BRKGA-QL(run_time)
 2:   Initialize Q-Table values
 3:   Generate Pop with p vectors of random-keys
 4:   Evaluate and sort Pop by fitness
 5:   Store the best solution sol*
 6:   for run_time seconds repeat
 7:     Choose an action for each parameter from Q-Table
 8:     Create Pop+ elite set (P_e) using p_e as guide
 9:     Create Pop+ mutant set (P_m) using p_m as guide
10:     Create Pop+ offspring set (P_o) using ρ_e as guide
11:     Pop ← Pop+
12:     Evaluate and sort Pop by fitness
13:     if the best solution improved then
14:       Store the best solution sol*
15:       Set reward (r = 1) and update Q-Table
16:     if exploration or stagnation then
17:       Identify communities in P_e with LP method
18:       Apply local search in these communities
19:   return sol*
20: end procedure
```

The evolutionary process of BRKGA-QL (Lines 8-12) is the same as the classical BRKGA. The population is partitioned into two groups (Elite and Non-Elite), the Elite partition is copied to the next population, $p * p_m$ new random-key vectors (mutants) are randomly created, and the offspring is created with *parametric uniform crossover* [19]. The new chromosomes are evaluated by their specific decoder and the new population is sorted by fitness.

At Lines 13-15, if the selected parameter configuration provides an improvement in the best chromosome of the population, it will gain a reward value of 1, and the function $Q(s, a)$ is updated by the Equation 1 for each pair s, a of the selected configuration.

The local search module proposed by [2] is also used in the BRKGA-QL (Lines 16-18). This component is called when $r = 1$ (exploration) or when the best chromosome is not improved for some generations (stagnation). Communities are identified in the Elite partition using the Label Propagation (LP) method [14] and the local search heuristics are applied only in the best chromosome of each community identified by LP that has not yet explored. The solution of the local search is not added to the population, but it is stored as the best solution (sol^*) if there is an improvement.

The BRKGA-QL proposed to solve the TSP has four local search heuristics. The order in which they are used may impact the quality of the obtained solution, hence it needs to be configured. For this reason, a variation of the Variable Neighborhood Descent (VND) [8] that does not have a predefined order for the heuristics was used. It is called Random Variable Neighborhood Search (RVND)

[11], and works by randomly selecting the neighborhood heuristic order to be applied at each iteration. Algorithm 4 shows the pseudocode of the RVND.

Algorithm 4 Random VND

```
1: procedure RVND(sol)
2:   Initialize the Neighborhood List (NL)
3:   while NL ≠ 0 do
4:     Choose a neighborhood N^i ∈ NL at random
5:     Find the best neighbor sol' of sol ∈ N^i
6:     if sol' is better than sol then
7:       sol ← sol'
8:       Restart NL
9:     else
10:      Remove N^i from the NL
11:  return sol
12: end procedure
```

The BRKGA-QL uses as stopping criteria the algorithm running time (in seconds) defined by the user (run_{time}) since the number of generations becomes less important due to the adaptive population size. This is a more user-friendly parameter, based on available computational resources and the complexity of the decoders.

3.1 Decoders

For the TSP, five decoders were developed to perform the mapping between the random-key solutions and the TSP solutions. All functions start from the ordering of the random-key vector in ascending order. With this sequence, a constructive or refinement heuristic is applied to obtain the TSP solution.

The decoders are:

1. Standard: the route is the sequence obtained by ordering the random-key vector;
2. *2-Opt*: from the sequence obtained by ordering the random-key vector, one full iteration of the 2-Opt heuristic is applied using the first improvement strategy;
3. *Cheapest Insertion*: from the sequence obtained by ordering the random-key vector, a sub-route is created with the first three customers and, from the fourth customer in the sequence, the cheapest position is calculated to insert the current customer. The cost of inserting a customer k after customer j is calculated by $c_{k,j} = d_{j,k} + d_{k,j+1} - d_{j,j+1}$;
4. *k-Farthest Insertion*: from the sequence obtained by ordering the random-key vector, a sub-route is created with the first customer in the sequence. In each iteration, one finds the customer i, among the first k customers in the sequence, which is the farthest from the current partial route. This customer

i is then inserted into the route at the position with the cheapest insertion cost. This step is repeated until all customers are inserted in the route;
5. *k-Nearest Insertion*: the procedure of this decoder is similar to the previous one. But it searches for the customer i, among the first k customers in the sequence, which is the nearest to the current partial route.

3.2 Local Searches

The RVND method randomly selects the order of neighborhoods to be applied in each iteration. For TSP, four classic local search heuristics from the literature were implemented:

1. 2-Opt [9]: Two non-adjacent arcs are deleted and another two are added in such a way that a new route is generated;
2. Or-opt2 [9]: Two adjacent customers are removed and inserted in another position of the route;
3. Node-Exchange [15]: Permutation between the positions of two customers.
4. Node-Insertion [15]: One customer is removed and inserted in another position of the route.

4 Computational Results

To evaluate the efficiency and robustness of the proposed method, computational experiments were carried out with 50 classic instances of the TSP obtained from TSPLib [4]. The size of the instances ranges from 51 to 3795 cities. The BRKGA-QL stopping criteria (run_{time}) was defined as the number of cities in the instance. Thus, it was possible to carry out the experiments without the need to configure this parameter. Each instance was executed 20 times with different random seeds.

BRKGA-QL was implemented in the C++ language and the framework will be available in the GitHub repository for free access (`https://github.com/antoniochaves19/BRKGA-QL`). The computational tests were performed on the workstation with two Intel Xeon Silver 4114 processors, each with 20 threads, 96 GB of RAM, and CentOS Linux 8.2 operating system. BRKGA-QL makes use of parallelism using the OpenMP paradigm.

The performance of the BRKGA-QL was measured by the relative percentage deviation (RPD) defined by Equation :

$$RPD(\%) = \frac{(sol - Optimal)}{Optimal} \times 100 \tag{2}$$

in which, *sol* is the solution value found by the BRKGA-QL and *Optimal* is the value of the optimal solution available in the TSPLib. Table 2 reports the computational results for each instance. The first and second columns show the instance name and number of cities, respectively. The columns RPD_b and RPD_a show the RPD obtained from the best and the average solutions of the

BRKGA-QL, respectively. The column SD shows the relative standard deviation (in percentage). The columns T^* and T show the average computational time to find the best solution and the complete time to run, in seconds.

As can be observed in Table 2, the BRKGA-QL found the optimal solution in 36 of the 50 tested instances. The average RPD_b was 0.14% and the worst value was less than 1%. BRKGA-QL was able to find good solutions in a few seconds for instances up to 200 cities. The average time to find the best solution was less than 50% of the total computational time. The RPD_a and the SD values were also close for the optimal solutions. Figure 5 shows the box plots of the RPD_b and RPD_a, the medians of the BRKGA-QL are similar ($RPD_b = 0.00$ and $RPD_a = 0.015$), but the RPD_b has a small interquartile range (RPD_b 1st quarter = 0.00, 3rd quarter = 0.03, and RPD_a 1st quarter = 0.00, 3rd quarter = 0.37).

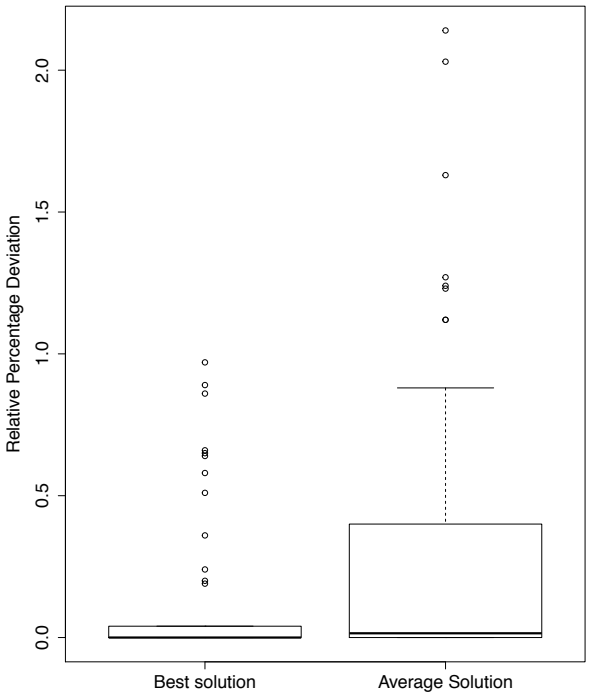

Fig. 5. Box plot of the RPD with best and average solutions of BRKGA-QL applied to TSP.

Figures 6 and 7 illustrate the behavior of BRKGA-QL to solve the TSP. It can be seen in Figure 6 that the method was able to find the optimal solution (known in the literature) for all instances up to 200 cities and in larger instances, the difference between the best solution found and the optimal solution (*gap*) was below 1%, in some instances the optimal was also found. Figure 7 shows

Table 2. Computational results of BRKGA-QL applied to TSP instances.

Instance	*#cities*	*Optimal*	RPD_b	RPD_a	*SD*	T^*	T
eil51	51	426	0.00	0.00	0.00	2.09	51.02
berlin52	52	7542	0.00	0.00	0.00	0.01	52.03
st70	70	675	0.00	0.00	0.00	0.04	70.02
eil76	76	538	0.00	0.00	0.00	1.70	76.03
pr76	76	108159	0.00	0.00	0.00	0.08	76.03
rat99	99	1211	0.00	0.00	0.00	0.50	99.03
kroA100	100	21282	0.00	0.00	0.00	0.05	100.04
kroB100	100	22141	0.00	0.14	0.14	15.77	100.03
kroC100	100	20749	0.00	0.00	0.00	0.25	100.04
kroD100	100	21294	0.00	0.00	0.00	0.31	100.03
kroE100	100	22068	0.00	0.00	0.00	8.97	100.05
rd100	100	7910	0.00	0.00	0.00	0.18	100.03
eil101	101	629	0.00	0.01	0.01	6.39	101.04
lin105	105	14379	0.00	0.00	0.00	0.06	105.02
pr107	107	44303	0.00	0.00	0.00	0.11	107.05
pr124	124	59030	0.00	0.00	0.00	0.07	124.05
bier127	127	118282	0.00	0.04	0.04	27.04	127.04
ch130	130	6110	0.00	0.00	0.00	1.80	130.06
pr136	136	96772	0.00	0.00	0.00	5.23	136.05
pr144	144	58537	0.00	0.00	0.00	0.30	144.08
ch150	150	6528	0.00	0.30	0.30	26.17	150.07
kroA150	150	26524	0.00	0.00	0.00	7.00	150.06
kroB150	150	26130	0.00	0.01	0.01	14.56	150.06
pr152	152	73682	0.00	0.00	0.00	0.24	152.10
u159	159	42080	0.00	0.00	0.00	0.13	159.09
rat195	195	2323	0.00	0.40	0.40	102.39	195.12
d198	198	15780	0.00	0.07	0.07	42.24	198.16
kroA200	200	29368	0.00	0.12	0.12	89.64	200.13
kroB200	200	29437	0.00	0.02	0.02	69.44	200.14
ts225	225	126643	0.00	0.00	0.00	1.72	225.16
tsp225	225	3916	0.66	1.12	0.46	83.27	225.20
pr226	226	80369	0.00	0.00	0.00	10.90	226.16
gil262	262	2378	0.04	0.28	0.23	83.28	262.23
pr264	264	49135	0.00	0.03	0.03	47.94	264.26
a280	280	2579	0.00	0.25	0.25	134.66	280.42
pr299	299	48191	0.00	0.14	0.14	173.80	299.29
lin318	318	42029	0.24	0.88	0.65	154.09	318.26
rd400	400	15281	0.20	0.85	0.65	238.07	400.50
fl417	417	11861	0.00	0.05	0.05	152.19	417.62
pr439	439	107217	0.19	0.46	0.27	258.39	439.54
d493	493	35002	0.58	1.12	0.54	322.86	493.74
u574	574	36905	0.51	1.24	0.73	375.00	575.29
rat575	575	6773	0.89	2.14	1.24	452.40	576.06
p654	654	34643	0.01	0.09	0.08	411.38	655.29
d657	657	48912	0.65	1.27	0.62	503.67	658.87
pr1002	1002	259045	0.97	2.03	1.06	799.36	1006.50
u1060	1060	224094	0.86	1.63	0.76	949.04	1064.55
vm1084	1084	239297	0.64	1.23	0.59	791.50	1087.35
fl1400	1400	20127	0.36	0.66	0.29	1008.216	1407.60
pr2392	2392	378032	0.00	0.00	0.00	276.61	2421.18
			0.14	0.33	0.20	153.02	337.17

that the deviation between the best and average solution found of the 20 runs is also low (close to 1% in the worst cases). These computational results show that the BRKGA-QL finds good results for the TSP.

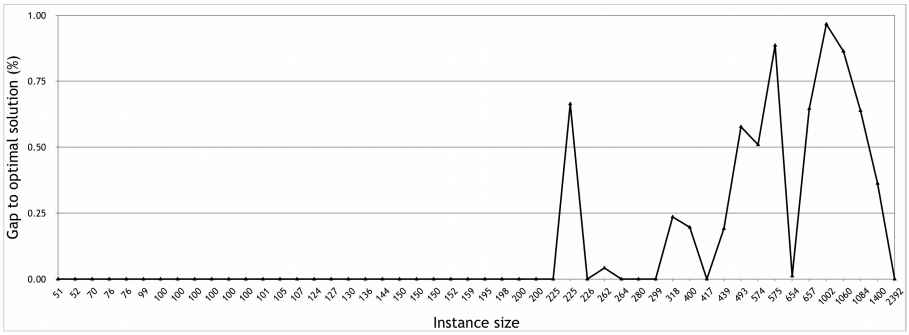

Fig. 6. Relative difference between the optimal solution and the best solution found by the BRKGA-QL for TSP instances.

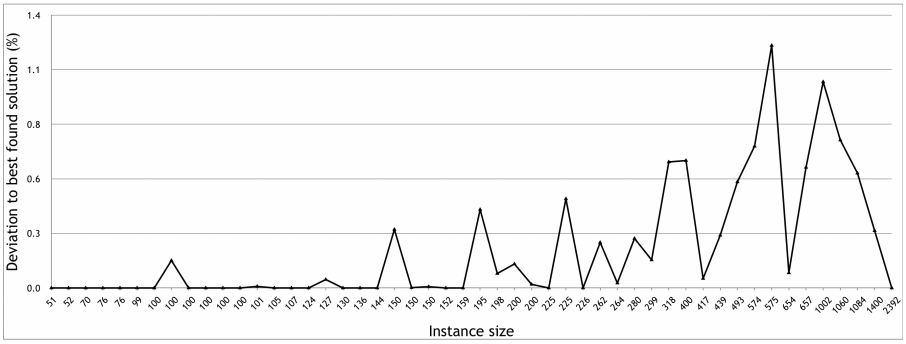

Fig. 7. Relative difference between the best solution and the average solution found by the BRKGA-QL for TSP instances.

Figure 8 presents the behavior of the rewards accumulated during a BRKGA-QL run. There is an accelerated growth at the beginning of the evolutionary process and, during the next generations, fewer rewards are earned. Such behavior was expected due to the policy of only awarding a reward ($r = 1$) when there is an improvement in the best solution found so far. This feature will be the subject of further studies to avoid spacing out the awarding of rewards over the generations.

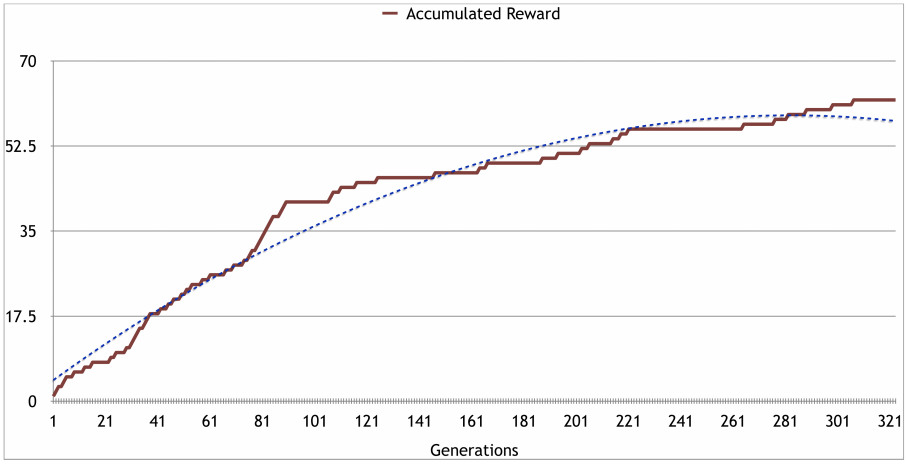

Fig. 8. Example of a reward curve accumulated during a BRKGA-QL run.

A general analysis of Figures 9 and 10 demonstrate the performance of BRKGA-QL when compared to classic versions of BRKGA, with and without local search module, and using the standard decoder and parameters tuned by iRace [7]. We use values shown in Table I as input of iRace and ten instances randomly selected. The best configuration of BRKGA found by iRace was $p = 987$, $p_e = 0.20$, $p_m = 0.15$, and $\rho_e = 0.70$. The median RPD_a of BRKGA-QL was 0.015% while the median RPD_a of BRKGA without local search was 67.84% and with local search was 0.34%. The difference between the maximum values was even more expressive, BRKGA-QL was 2.14% and BRKGA without local search was 1968%. BRKGA with local search has a behavior near to BRKGA-QL, but the maximum value was 6.71%.

Figure 11 shows the performance profile of BRKGA-QL and BRKGA with local search. We use the performance profile as a convergence test to evaluate the level of accuracy of the approaches. The tolerance was set to 1% of the RPD_a, i.e., the computational time of results with RPD_a equal or greater than 1% was set to ∞ to indicate that the approach fails to satisfy the convergence test for the respective instance. We can observe that BRKGA-QL presents the best results in terms of total computational time and level of accuracy. For the level of accuracy defined, BRKGA-QL solves close to 84% of the instances, while BRKGA solves 62% of the instances. We also noted in the plot that BRKGA-QL is the fastest heuristic in 65% of the instances, while BRKGA is the fastest in 18% of the instances, in terms of computational time to find the best solution. The performance profile also shows that BRKGA requires more than twice the computational time as BRKGA-QL on 44% of the instances (with performance ratio = 2, BRKGA = 32% and BRKGA-QL = 76%).

The Wilcoxon signed-rank test (WSR) [16], a non-parametric statistical hypothesis test, was used to compare the quality of the solution of BRKGA-QL and

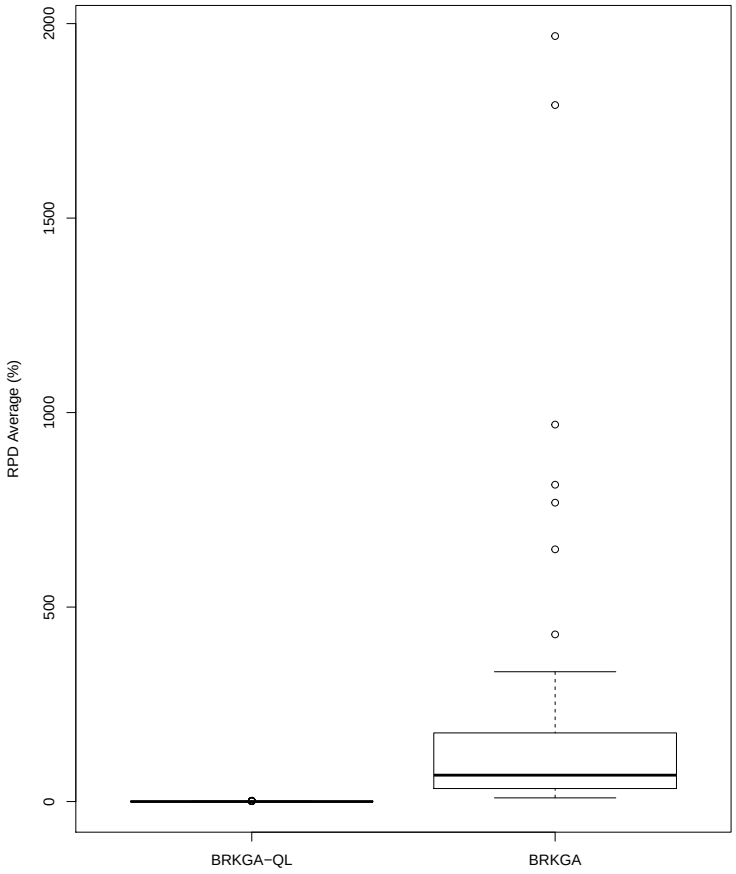

Fig. 9. Box plot of the RPD_a with average solutions of BRKGA-QL and BRKGA without local search module.

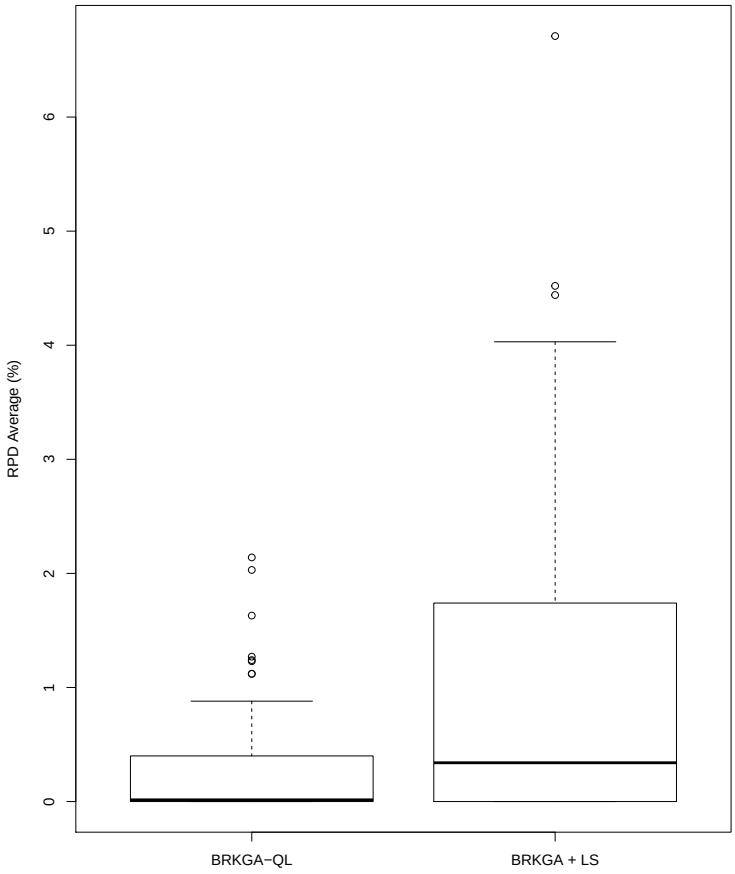

Fig. 10. Box plot of the RPD_a with average solutions of BRKGA-QL and BRKGA with local search module.

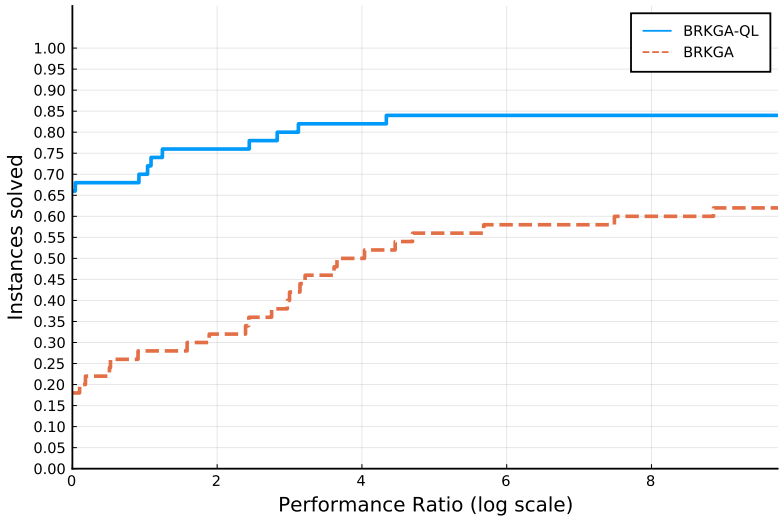

Fig. 11. Performance profile of runtime for BRKGA-QL and BRKGA with local search

the BRKGA variants (with/without local search). The WSR indicated that there is a statistically significant difference between BRKGA-QL and the BRKGA without local search ($p\text{-}value = 3.89e\text{-}10$) and BRKGA with local search ($p\text{-}value = 2.57e\text{-}07$). These results suggest that the use of the Q-Learning method, different decoders, and local search heuristics on the proposed method found better solutions in terms of quality.

5 Discussion and future works

This research paper presented a Reinforcement Learning method to online control the BRKGA parameters. The Q-Learning method was used to provide the best configuration of parameters for a specific tested instance. As the generations of BRKGA-QL are being executed, Q-Table becomes increasingly efficient due to the obtained knowledge.

The computational results show that the BRKGA-QL found good solutions for the Traveling Salesman Problem (TSP). The proposed method was able to find the optimal solution in 72% of the TSP instances tested, with small relative percentage deviations for the best and the average solutions.

The BRKGA-QL was applied only in the context of the symmetric TSP. However, other optimization problems can be easily solved by the BRKGA-QL with the development of new decoders and local search heuristics.

For future works, it is interesting to perform computational tests with other problems, and an analysis of different policies to update the Q-Table can be conducted to allow a more efficient response.

Acknowledgment

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, by the FAPESP (Grant No. 2018/15417-8), and by Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq (Grant No. 303736/2018-6, 423694/2018-9).

References

1. Bean, J.C.: Genetic algorithms and random keys for sequencing and optimization. ORSA Journal on Computing **6**(2), 154–160 (1994)
2. Chaves, A.A., Gonçalves, J.F., Lorena, L.A.N.: Adaptive biased random-key genetic algorithm with local search for the capacitated centered clustering problem. Computers & Industrial Engineering **124**, 331 – 346 (2018)
3. Evans, B., Xue, B., Zhang, M.: An adaptive and near parameter-free evolutionary computation approach towards true automation in automl. In: 2020 IEEE Congress on Evolutionary Computation (CEC). pp. 1–8 (2020). https://doi.org/10.1109/CEC48606.2020.9185770
4. Gerhard, R.: Tsplib 95 documentation. University of Heidelberg (1995)
5. Goncalves, J.F., Resende, M.G.C.: Biased random-key genetic algorithms for combinatorial optimization. Journal of Heuristics **17**(5), 487–525 (2011)
6. Kizilay, D., Tasgetiren, M.F., Oztop, H., Kandiller, L., Suganthan, P.: A differential evolution algorithm with q-learning for solving engineering design problems. In: 2020 IEEE Congress on Evolutionary Computation (CEC). pp. 1–8. IEEE (2020)
7. López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L.P., Birattari, M., Stützle, T.: The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives **3**, 43–58 (2016)
8. Mladenovic, N., Hansen, P.: Variable neighborhood search. Computers and Operations Research **24**, 1097–1100 (1997)
9. Or, I.: TRAVELING SALESMAN TYPE COMBINATORIAL PROBLEMS AND THEIR RELATION TO THE LOGISTICS OF REGIONAL BLOOD BANKING. Ph.D. thesis, Northwestern, USA (1976)
10. Öztop, H., Tasgetiren, M.F., Kandiller, L., Pan, Q.K.: A novel general variable neighborhood search through q-learning for no-idle flowshop scheduling. In: 2020 IEEE Congress on Evolutionary Computation (CEC). pp. 1–8. IEEE (2020)
11. Penna, P.H.V., Subramanian, A., Ochi, L.S.: An iterated local search heuristic for the heterogeneous fleet vehicle routing problem. Journal of Heuristics **19**(2), 201–232 (2013)
12. Prasetyo, H., Fauza, G., Amer, Y., Lee, S.: Survey on applications of biased-random key genetic algorithms for solving optimization problems. In: Industrial Engineering and Engineering Management (IEEM), 2015 IEEE International Conference on. pp. 863–870. IEEE (2015)
13. Queiroz dos Santos, J.P., de Melo, J.D., Duarte Neto, A.D., Aloise, D.: Reactive search strategies using reinforcement learning, local search algorithms and variable neighborhood search. Expert Systems with Applications **41**(10), 4939 – 4949 (2014). https://doi.org/https://doi.org/10.1016/j.eswa.2014.01.040, `http://www.sciencedirect.com/science/article/pii/S0957417414000645`
14. Raghavan, U.N., Albert, R., Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. Physical Review E **76**(3), 036106 (2007)

15. Rego, C., Glover, F.: Local search and metaheuristics for the travelling salesman problem. Gutin and AP Punnen (eds.), The Travelling Salesman Problem and its Variations (2002)
16. Rey, D., Neuhauser, M.: Wilcoxon-signed-rank test. In: Lovric, M. (ed.) International Encyclopedia of Statistical Science, pp. 1658–1659. Springer Berlin Heidelberg (2014)
17. Samma, H., Mohamad-Saleh, J., Suandi, S.A., Lahasan, B.: Q-learning-based simulated annealing algorithm for constrained engineering design problems. Neural Computing and Applications **32**(9), 5147–5161 (2020)
18. Sevaux, M., Sörensen, K., Pillay, N.: Adaptive and multilevel metaheuristics. Handbook of Heuristics pp. 1–19 (2018)
19. Spears, W.M., Jong, K.A.D.: On the virtues of parameterized uniform crossover. Proc. of the Fourth International Conference on Genetic Algorithms pp. 230–236 (1991)
20. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
21. Talbi, E.G.: Metaheuristics: from design to implementation, vol. 74. John Wiley & Sons (2009)
22. Watkins, C.J.C.H.: Learning from delayed rewards. Ph.D. thesis, King's College, University of Cambridge (1989)
23. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE transactions on evolutionary computation **1**(1), 67–82 (1997)