Fairer Comparisons for Travelling Salesman Problem Solutions Using Hash Functions

Mehdi El Krari, Rym Nesrine Guibadj, John Woodward, Denis Robilliard

Fairer comparisons for Travelling Salesman Problem solutions using Hash Functions

Mehdi El Krari[1][0000−0001−7936−9430], Rym Nesrine Guibadj[2][0000−0003−3448−3912], John Woodward[3][0000−0002−2093−8990], and Denis Robilliard[2][0000−0003−1836−6679]

[1] Computational Optimisation and Learning Lab, School of Computer Science, University of Nottingham, Nottingham, UK
mehdi@elkrari.com
[2] Université du Littoral Côte d'Opale, EA 4491 - LISIC, Calais, France
{rym.guibadj,denis.robilliard}@univ-littoral.fr
[3] Operational Research Group, School of Electronic Engineering and Computer Science, Queen Mary University of London, Mile End Road, London E1 4NS, UK
j.woodward@qmul.ac.uk

Abstract. Fitness functions fail to differentiate between different solutions with the same fitness, and this lack of ability to distinguish between solutions can have a detrimental effect on the search process. We investigate, for the Travelling Salesman Problem (TSP), the impact of using a hash function to differentiate solutions during the search process. Whereas this work is not intended to improve the state-of-the-art of the TSP solvers, it nevertheless reveals a positive effect when the hash function is used.

Keywords: Hash functions · Combinatorial Problems · Travelling Salesman Problem · Local Search · Genetic Algorithms · Memetic Algorithms

1 Introduction

The way a solution of a Combinatorial Optimisation Problem (COP) can be represented is a key issue to design an efficient search algorithm to solve it. A representation associates an encoding, that can be easily evaluated during the search algorithm. For example, if we consider the Travelling Salesman Problem (TSP), a solution to this problem is a tour in which all the cities are listed in the order they are visited, and each city is visited only once. This solution can be represented using different encodings [8, 16]: binary, graphs, permutations etc. In the permutation representation of the TSP, this is interpreted as a sequence of cities in which the first and the last elements are connected. The cost of a sequence depends on the order of the cities in the permutation.

The permutation is used as a solution encoding in many other COPs. They can be found in various application areas such as assignment problems [11, 14, 17], scheduling problems [2], routing problems [17], etc.

An evaluation function, that associates a fitness measure to each solution, should be defined in order to (i) distinguish two solutions based on their quality and (ii) guide the search process. Often these two purposes are expected to be met by a single function [1].

The mapping from solutions space to fitness values may belong to one of the following cases:

- 1-to-1 mapping: each fitness value is associated to only one solution
- n-to-1 mapping: several different solutions have the same fitness value

While a canonical form was proposed in genetic programming by Woodward [21], it still not evident to find it for many COPs. It is common for a fitness function to map different solutions to the same fitness value. This means the metaheuristic cannot distinguish solutions based solely on their fitness values, and this loss of information may impede the search ability of the metaheuritics.

When the search space, i.e. the set of all the feasible solutions for a given instance, has many solutions with the same fitness value, this often results in large regions containing *plateaus*. A metaheuristic may repeatedly return to recently visited solutions as it wanders around the plateau as the fitness function does not provide any helpful information. We say a cycle occurs when the search process returns to an already visited solution again. This term is mentioned in the literature [3, 9] to describe the same phenomena. The problem of *cycling* may lead the metaheuristic to be confined to a particular area of the search space.

Another issue arising with population-based search techniques, such as Genetic Algorithms, is the premature convergence of the metaheuristic when different solutions have the same fitness in the last generations. Indeed, convergence measures are mainly based on population diversity to terminate the evolution. Usually, the diversity of the population is measured by assessing the similarity among solutions based on their fitness. One of the definitions of convergence in an evolutionary process is when a certain percentage of the population has the same fitness, thus indicating that the evolutionary process has stagnated [12].

Differentiating solutions by the mean of their respective fitness values is motivated by the low complexity induced by the comparison. It can even be constant ($\mathcal{O}(1)$) for some COPs, such as the TSP. On the other hand, differentiating solutions by their respective encoding (permutations, binary strings, etc.) is entirely accurate but more expensive. Comparing two permutations, for example, is linear ($\mathcal{O}(n)$), which can likely increase the complexity of the whole metaheuristic from $\mathcal{O}(n^k)$ to $\mathcal{O}(n^{k+1})$.

We introduced in a previous short paper [5] a new hash function for the TSP. In this study, we show its positive effect to provide relevant information during the search process. Experiments are conducted on the TSP but point to possible use on other COPs. Three metaheuristics are analysed: Iterated Local Search (ILS), Genetic Algorithms (GAs) and Memetic Algorithms (MAs). In this paper, we refer to solutions comparison as the differentiation mechanism to distinguish between two solutions.

The remainder of this paper is organised as follows. A formal definition of TSP is provided in the next section, with an analysis of the fitness values distri-

bution of some TSP instances over the search space. Section 3 introduces a new hash function designed for TSP permutations and gives a comparative study based on the number of collisions. Computational results are then presented in Section 4 to show the effect of the hash function on three metaheuristics. Finally, Section 5 presents our conclusions and our plans for future work.

2 Collision analysis of the fitness function on the TSP

The TSP is frequently used as a test-bed for designing effective methods to solve general sequencing permutation problems. The problem is modelled with a graph $G = (V, A)$ where $V = \{v_1, ..., v_n\}$ is the vertex set, and $A = \{(v_i, v_j)|v_i, v_j \in V, i \neq j\}$ is the edge set. A non-negative cost (or distance) matrix $C = (c_{ij})$ is associated with A. This paper focuses on the most widely studied form of the problem in which costs are assumed to be symmetric $c_{ij} = c_{ji}$ and satisfy the triangle inequality ($c_{ij} + c_{jk} > c_{ik}$). A feasible TSP solution is a sequence of nodes/cities arranged in a permutation π of size n. Its cost is the sum of the distances of each couple of adjacent cities in the permutation. We define a fitness function to evaluate a TSP permutation π as follows:

$$f_{\text{fit}}(\pi) = \sum_{i=1}^{n-1} c_{\pi_i \pi_{i+1}} + c_{\pi_n \pi_1} \tag{1}$$

It is common practice in evolutionary computation to use the fitness function, f_{fit}, to compare solutions. It is well-known that many solutions may map to the same fitness value, but, to the best of our knowledge, no prior work sheds light on how much the fitness values are repeated over a search space nor how they are distributed over its solutions. To do so, we chose 39 instances from the TSP benchmark TSPLIB [18] (sizes n range from 51 to 575). We then explore the search space in two different ways. Firstly, a set S_{LO} is composed of n^2 local optima obtained with an ILS framework to get as many neighbouring solutions as possible. Secondly, a set S_{rand} is built, containing $10 \times n^2$ random solutions. These samples were generated in such a way that all the solutions are distinct. This means that they do not contain two identical permutations. We compute in the first part of this section the number of collisions occurring in each sample for each instance.

We say that we have a collision between two solution s_1, s_2 if $f(s_1) = f(s_2)$, where the function f outputs a numeric value. We then examine how these collisions are distributed over the fitness values in Section 3.2. As an example, if 4 solutions s_1, s_2, s_3, s_4 map to the same value, we count 4 *repetitions* ($f(s_1) = f(s_2) = f(s_3) = f(s_4)$), and 6 *collisions* ($(s_1, s_2), (s_1, s_3), (s_1, s_4), (s_2, s_3), (s_2, s_4), (s_3, s_4)$). Thus the number of collisions may exceed the sample size.

2.1 Too many collisions for the fitness function

To determine if the fitness function as a comparison tool can affect a metaheuristic, we measure the collisions over the above-mentioned samples and list them in table 1. For each instance, we expose the sample size, $|S_{\text{LO}}|$ and $|S_{\text{rand}}|$, and the number of collisions, C_{LO} and C_{rand}, computed by comparing all the solution pairs (these values are rounded at $1E3$ — precise values are displayed in table 2); then the number of the different fitness values, Fit_{LO} and Fit_{rand}, retrieved in each sample.

Getting collisions from large samples of solutions is not surprising, especially when it comes to local optima that share common edges between them. But the number of collisions we have in table 1 exceeds our expectations. Indeed, a very high number of collisions is observed in almost all samples, with up to millions of collisions for the smallest ones. Moreover, according to Fit_{LO} and Fit_{rand}, we notice very small sets of fitness values to whom the solutions of S_{LO} and S_{rand} are mapping. In other words, the large set of solutions is distributed over a small set of fitness values, making some fitness values very repetitive.

Table 1: Collision analysis of the fitness function on the TSP

Instance	$\lvert S_{\text{LO}}\rvert$ 1E3	C_{LO} 1E3	Fit_{LO}	$\lvert S_{\text{rand}}\rvert$ 1E3	C_{rand} 1E3	Fit_{rand}	Instance	$\lvert S_{\text{LO}}\rvert$ 1E3	C_{LO} 1E3	Fit_{LO}	$\lvert S_{\text{rand}}\rvert$ 1E3	C_{rand} 1E3	Fit_{rand}
eil51	3	127	53	26	1,069	577	pr144	21	59	5,889	207	200	98,182
berlin52	3	5	799	27	65	6,833	ch150	23	478	893	225	3,921	10,792
st70	5	230	99	49	1,877	1,170	kroA150	23	129	2,807	225	707	46,224
eil76	6	456	72	58	4,140	766	kroB150	23	136	2,708	225	680	47,674
pr76	6	3	3,788	58	18	43,331	pr152	23	73	4,750	231	197	117,041
gr96	9	11	3,872	92	71	49,124	u159	25	71	5,647	253	530	70,958
rat99	10	457	189	98	3,317	2,613	rat195	38	4,736	310	380	25,839	5,329
kroA100	10	31	2,163	100	170	32,619	d198	39	1,338	1,047	392	3,001	38,231
kroB100	10	33	2,028	100	173	32,317	gr202	41	405	3,256	408	3,439	36,831
kroC100	10	30	2,168	100	169	32,881	ts225	51	128	12,854	506	825	170,884
kroD100	10	31	2,034	100	186	30,723	gr229	52	178	10,401	524	922	167,730
kroE100	10	34	1,970	100	168	33,161	gil262	69	16,165	308	686	96,690	4,930
rd100	10	71	1,025	100	599	12,060	a280	79	14,369	414	784	90,807	6,746
eil101	10	1,376	80	102	10,622	975	lin318	101	2,219	3,951	1,011	10,188	78,223
lin105	11	59	1,443	110	289	26,055	rd400	160	18,439	1,398	1,600	81,554	29,261
pr107	11	41	2,549	114	58	73,334	fl417	174	28,258	1,424	1,739	31,432	80,618
pr124	15	37	4,028	154	116	82,802	gr431	186	2,027	15,233	1,954	34,979	91,416
bier127	16	18	7,199	161	187	68,143	pcb442	196	7,548	4,649	1,858	7,753	303,216
ch130	17	351	734	169	2,419	9,609	rat575	331	234,671	592	3,306	676,190	16,898
gr137	19	37	5,593	188	179	89,176							

2.2 Distribution of collisions over fitness values

We count for each fitness value in each sample S_{LO} how many times it appears. We observed from the different gathered data a high similarity of distribution of repetitions between the different instances. For a better observation of these distributions, we draw for each instance a scatter plot with ascendant and linear scale axes, where each dot depicts the number of repetitions (Y-axis) of one fitness value (X-axis). The latter is illustrated by its gap (δ) from the optimal solution's fitness, calculated with the formula 2.

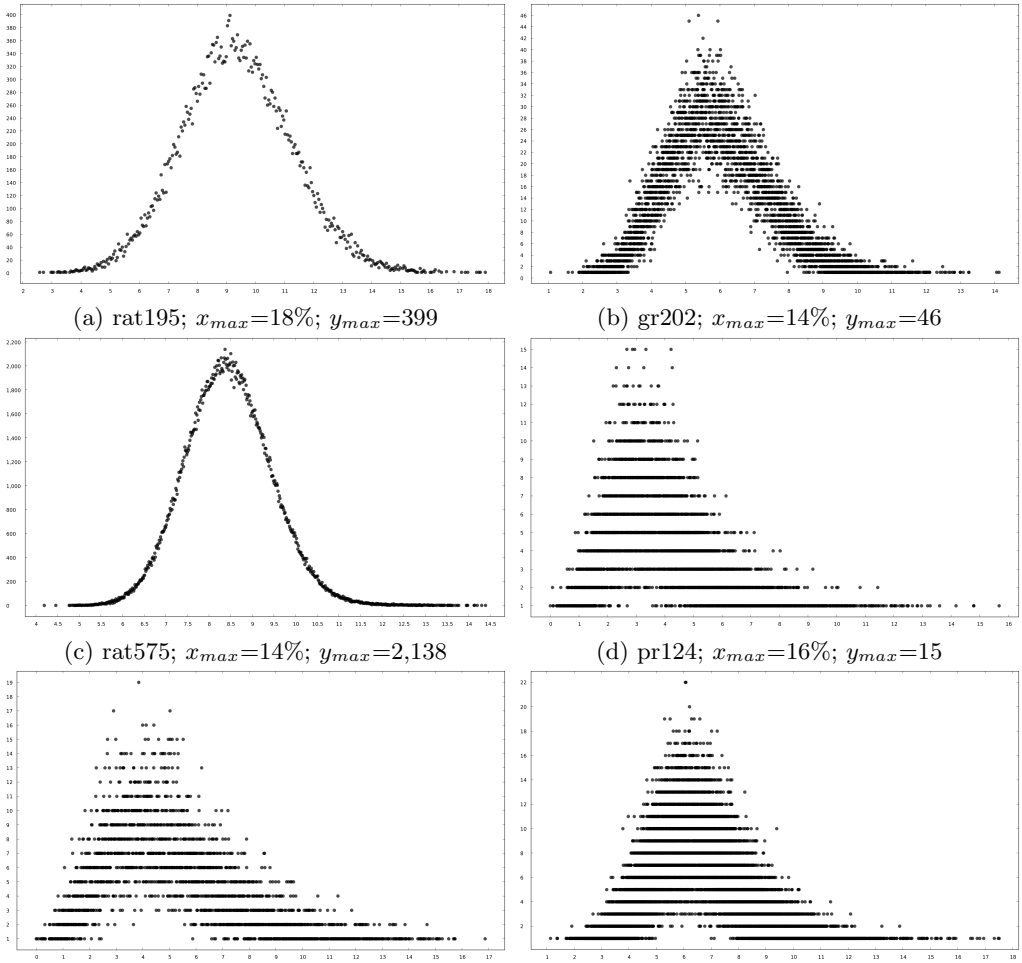
$$\delta = \frac{f_{\mathrm{fit}}(solution) - f_{\mathrm{fit}}(optimal)}{f_{\mathrm{fit}}(optimal)} \times 100(\%) \tag{2}$$

As expected, the similarity of distributions induces similarly shaped plots. It allowed us to classify them into two main distributions, represented in figure 1. Below each exposed plot in the figure, we provide (i) the instance name; (ii) the maximal value of the abscissa axis, i.e. the gap between the highest fitness value existing in S_{LO} and the optimal solution (x_{max}); (iii) the maximal value of the ordinate axis, which is the most important repetition observed in S_{LO} (y_{max}).

In 19 instances (from the 39 studied), we observed the distribution of repetitions with a bell curve. The first row of plots in Figure 1 shows 3 examples of these instances where we can notice a distribution close to normality. While the different x_{max} values are in a fairly narrow range and don't depend on the collisions caused by f_{fit}, there is a strong correlation between the density of the plots and their respective y_{max} value: the larger the value of y_{max}, the smaller the thickness of the curve will be and vice versa. To illustrate this correlation better, we put in our examples a pair of instances with approximately equal sizes (rat195 and gr202), where the scatter plot becomes more sparse when y_{max} decreases.

For instances with a low y_{max} (which doesn't mean a low number of collisions), the repetitions become more sparse on the plot until we move away from the normal distribution. The second row of the figure exposes 3 examples of the 17 instances where the repetitions form an area with a shape close to a bell. Unlike the first class of instances, the number of distinct fitness values is more important but each one appears in the sample less frequently.

Inspecting the collisions occurring in large samples of solutions reveals a high number of repetitions of the fitness values. This can mislead metaheuristics when the fitness function is used for comparing solutions. The analysis of the distribution of fitness repetitions unveils two major classes of distribution. A first one where instances have a few (distinct) fitness values but with high repetition. Then a second one with a reversed tendency. Such information can be exploited to predict how a trajectory-based metaheuristic can be influenced. For example, in a tabu search context, a high number of repetitions of fitness values may lead to short cycles when the algorithm considers each visited solution by its fitness value. These hypotheses are verified and validated in Section 4.



(a) rat195; x_{max}=18%; y_{max}=399

(b) gr202; x_{max}=14%; y_{max}=46

(c) rat575; x_{max}=14%; y_{max}=2,138

(d) pr124; x_{max}=16%; y_{max}=15

(e) kroC100; x_{max}=17%; y_{max}=19

(f) gr229; x_{max}=18%; y_{max}=22

Fig. 1: Distribution of fitness repetitions and their gaps from the optimal solution

3 Hash functions for a reliable comparison

The analysis shown in the previous section is a strong motivation to search for an efficient alternative to the objective-based fitness function to compare solutions. Hash functions for COPs (specifically permutation-based ones) have been proposed to yield a lower number of collisions than the given fitness function.

3.1 Existing hash functions

Woodruff and Zemel introduced three hash functions in [20] . The first function, h_1, is based on multiplying pseudo-random integers ρ_i with each element of

the solution vector π_i. The maximum integer value $MAXINT$ is used to avoid overflow. The second function, h_2, makes use of a matrix P of a pre-computed random weight, while the last function, h_3, replaces an entry $P(i,j)$ with $P(i) \times P(j)$ which is equivalent to replacing the matrix P with a long vector of pre-computed random weights. Since the authors claimed h_3 is better than h_2, the latter will not appear in our comparative study.

$$h_1 = (\sum_{i=1}^{n} \rho_i \pi_i) \% (MAXINT + 1) \tag{3}$$

$$h_2 = \sum_{i=1}^{n} P(\pi_i, \pi_{i+1}) \tag{4}$$

$$h_3 = (\sum_{i=1}^{n} P(\pi_i) P(\pi_{i+1})) \% (MAXINT + 1) \tag{5}$$

The three hash functions were designed taking into account the following properties:

1. Computation and update of the hash value should be as fast as possible, and in any case as fast as the fitness function. The hash value update after applying a move on a candidate solution should preferably be computed in $\mathcal{O}(1)$ time.
2. The hash values should be in a range that results in reasonable storage requirements and comparison effort.
3. The function should guarantee a low collision probability to minimise the risk of two permutations having identical hashes.

Toffolo et al. [19] employed the two hash functions defined in Eq. 6 and Eq. 7 to rapidly evaluate a newly explored route of the CVRP. The function h_p is a multiplicative hash which depends on the visited permutation. The second function, h_s, is an additive hash that depends on the set of visited customers. These hash functions were used with two different values of ρ. It is set to the prime number 31, or to the smallest prime number greater than the number of customers. To prevent overflow during multiplication, the values ρ^i were bounded taking the rest of the integer division by a large number. While a solution in a CVRP instance uses a subset of customers, it is not the case for the TSP. Using all the customers/nodes for any solution of the latter makes h_s have the same value for a given instance and thus cannot be used as a hash function for the TSP.

$$h_p(\pi) = \sum_{i=1}^{n} \rho^i \pi_i \tag{6}$$

$$h_s(\pi) = \sum_{i=1}^{n} \rho^{\pi_i} \tag{7}$$

3.2 The proposed function

In addition to the three properties (previously mentioned in Section 3.1) that a hash function should acquire, we propose in this paper to implement a hash function with an added characteristic.

While the existing functions are based on vectors of large random values, we want to design our hash function (η) with only the solution and the instance data already stored by the heuristic. This is more challenging since random values help to reduce the number of collisions considerably. Conversely, solution data can increase collisions due to the correlation and similarities we may observe between a pair of solutions. The fitness function, based on the distance matrix, is a concrete example.

To make sure η obeys the property n° 1, we define a (sub-)hash function η_e for one edge of the permutation. η can be formulated then as described in formula 8. It also ensures to get the same hash value when a permutation is shifted since the solution stays the same in the TSP case.

$$\eta = \eta_e(1, n) + \sum_{i=1}^{n-1} \eta_e(i, i+1) \tag{8}$$

The operands we chose for our hash function are the distance matrix and the set of node identifiers which are n distinct integers in the range $[1; n]$. In addition to the mathematical operators, we define a new operator *mod* in Formula 9. This definition is an adjustment of the classical modulo operator to ensure having the same hash value when the permutation is symmetrical and prevent the η_e function from returning a zero value ($(a < b) \Rightarrow (a\%b = 0)$).

$$mod(a, b) = max(a, b)\%min(a, b) \tag{9}$$

To lower the number of collisions, we favoured multiplication over addition since it gives more diverse values. The division is dismissed to avoid dealing with precision issues. We decided to involve more the node identifiers rather than the distance matrix. Values of the latter are larger and will quickly lead to memory overflows.

Following all the guidelines mentioned above, we designed the function η_e as shown in formula 10. π_i is the identifier of the i^{th} node in the permutation π. $C = (c_{ij})$ is the distance matrix of the studied instance. Formula 11 defines our hash function η.

$$\eta_e(i, j) = mod(\pi_i, \pi_j) \times (\pi_i + \pi_j) \times (\pi_i \times \pi_j) \times c_{\pi_i, \pi_j} \tag{10}$$

$$\begin{aligned}\eta &= \eta_e(1,n) + \sum_{i=1}^{n-1} \eta_e(i,i+1) \\ &= mod(\pi_1,\pi_n) \times (\pi_1 + \pi_n) \times (\pi_1 \times \pi_n) \times c_{\pi_1,\pi_n} + \\ &\sum_{i=1}^{n-1} mod(\pi_i,\pi_{i+1}) \times (\pi_i + \pi_{i+1}) \times (\pi_i \times \pi_{i+1}) \times c_{\pi_i,\pi_{i+1}}\end{aligned} \tag{11}$$

3.3 Comparative study

Table 2 below compares the fitness function f_{fit} and the hash functions η and h_3 (which was chosen as the best of the functions in Section 3.1 after a preliminary comparison [4]).

For each instance sample (S_{LO} and S_{rand}), the number of collisions resulting from each function is printed on the table.

The first remark is the significant reduction of collisions made between f_{fit} and the hash functions, which can't be with no effect on the search process. The second observation is the excellent results obtained by η compared with h_3, especially for large instances. Our designed hash function succeeded in getting zero collisions on 36 (resp. 32) instances for S_{LO} (resp. S_{rand}), against only 29 (resp. 10) instances for h_3. The overall average for η is less than one collision in each set of samples, while it is much higher for h_3 in S_{rand}.

This shows that it is possible to design a hash function with fewer collisions than those proposed in the literature.

η can then be embedded in a metaheuristic, with a constant time cost, as a reasonable alternative to fitness evaluation in order to compare solutions. In addition to the three properties listed in Section 3.1, this hash function only uses solution data and does not need large vectors of random values, thus reducing its memory footprint. Note that we did not encounter any overflow with our proposed hash function on our set of instances and samples. Nonetheless, in the event of overflow, one could use standard strategies, such as clipping values with a modulo operator (see also [19]).

The following section shows the multiple effects of using the hash functions.

4 Revisiting some metaheuristics with hash functions

Let's consider f_{comp} the comparison function to check the equality between a pair of solutions. The results of each test/run performed in this section are obtained with $\text{f}_{\text{comp}} = f_{\text{fit}}$, then $\text{f}_{\text{comp}} = \eta$. We provide the same input in each case. Fifty runs are assigned to each instance/test.

[4] The comparison between all the functions is available at https://elkrari.com/hashfunctions/

Table 2: A comparison of the number of collisions between f_{fit}, h_3 and η obtaines on the samples S_{rand} and S_{LO}

Instance	S_{LO}			S_{rand}			Instance	S_{LO}			S_{rand}		
	f_{fit}	η	h_3	f_{fit}	η	h_3		f_{fit}	η	h_3	f_{fit}	η	h_3
eil51	126,875	0	0	1,068,752	2	0	pr144	59,098	0	0	199,699	0	0
berlin52	5,495	0	0	64,662	0	0	ch150	477,971	0	0	3,921,003	0	4
st70	229,647	0	0	1,876,506	0	0	kroA150	128,766	0	0	707,170	0	4
eil76	456,422	1	0	4,140,456	2	1	kroB150	135,785	0	1	680,086	0	3
pr76	2,838	0	0	18,107	0	0	pr152	72,564	0	0	196,547	0	1
gr96	10,633	0	0	71,134	1	0	u159	70,763	0	0	529,831	0	2
rat99	457,303	0	0	3,317,137	2	0	rat195	4,736,380	0	0	25,838,895	1	5
kroA100	30,902	0	0	170,251	1	0	d198	1,337,616	0	0	3,001,233	0	10
kroB100	32,978	0	1	172,655	0	1	gr202	404,695	0	0	3,439,300	0	13
kroC100	30,277	0	0	169,127	0	1	ts225	128,149	0	0	824,742	0	19
kroD100	30,805	0	0	185,716	0	2	gr229	177,592	0	1	921,695	0	15
kroE100	33,600	0	0	167,732	0	1	gil262	16,165,285	0	0	96,690,038	0	26
rd100	71,443	0	0	598,845	0	1	a280	14,368,564	0	2	90,806,864	0	31
eil101	1,376,150	0	0	10,621,646	1	0	lin318	2,219,290	0	1	10,188,376	0	45
lin105	59,275	0	0	288,911	0	1	rd400	18,439,021	0	2	81,553,529	0	99
pr107	41,307	0	0	58,422	0	1	fl417	28,258,436	1	3	31,432,278	0	97
pr124	36,773	0	0	116,389	0	0	gr431	2,026,980	0	2	34,978,940	0	127
bier127	17,641	7	0	187,417	0	5	pcb442	7,548,302	0	3	34,978,940	0	138
ch130	350,821	0	1	2,419,326	0	4	rat575	234,671,318	0	4	676,190,119	0	382
gr137	37,181	0	0	179,311	0	3	average	8,586,280.54	0.23	0.54	28,794,148.38	0.26	26.72

This section doesn't aim to improve the state-of-the-art. The objective is to provide a comparison of the two scenarios of f_{comp} in the same environment. For each metaheuristic, we implement a basic version with known operators and strategies.

4.1 Cycling Analysis

One of the limitations of ILS is cycling. A cycle occurs when the search returns to an already visited local optimum, which means the algorithm is stuck in a limited region of the search space. To inspect the effect of using hash functions to identify cyclings, we ran an ILS with a stochastic local search [10] by the 2-Opt neighbourhood function [4, 7] and a perturbation of n 2-Opt random moves. Each run stops when the first cycle occurs or when the algorithm reaches $50 \times n$ iterations.

Table 3 shows the average number of visited solutions before a cycle arises. For each instance, we note the maximum number of iterations (Max_{iter}), which is also the maximum number of local optima we can visit in each run. For each case of f_{comp}, we record how many times a cycle happened (C), and the average number of visited local optima before ILS stops (or the history size $|H|$). We note zero when no cycle appears during the 50 runs. The last column of the

table compares the two cases of f_{comp} with the Mann Whitney U Test (also called Wilcoxon Rank Sum Test) [15, 6] The test measures the separation level between the number of iterations made in each case with a $p-$value.

Table 3: Cycling analysis of the ILS framework when using η then f_{fit} to differentiate solutions.

Instance	Max_{iter}	$f_{comp} = \eta$		$f_{comp} = f_{fit}$		$p-$value	Instance	Max_{iter}	$f_{comp} = \eta$		$f_{comp} = f_{fit}$		$p-$value
		C	$\|H\|$	C	$\|H\|$				C	$\|H\|$	C	$\|H\|$	
eil51	2550	22	1463.18	50	7.78	6.86E-18	pr144	7200	50	608.22	50	100.1	2.77E-16
berlin52	2600	50	401.8	50	39.96	6.86E-18	ch150	7500	**0**		50	35.54	6.86E-18
st70	3500	10	1946.5	50	11.98	6.86E-18	kroA150	7500	**0**		50	56.74	6.86E-18
eil76	3800	1	3733	50	8.32	6.86E-18	kroB150	7500	**0**		50	58.72	6.86E-18
pr76	3800	41	1990.22	50	110.32	6.86E-18	pr152	7600	4	6042.25	50	106.12	6.86E-18
gr96	4800	**0**		50	97.72	6.86E-18	u159	7950	**0**		50	95.2	6.86E-18
rat99	4950	2	3054.5	50	11.56	6.86E-18	rat195	9750	**0**		50	15.6	6.86E-18
kroA100	5000	14	3115.64	50	58.24	6.86E-18	d198	9900	**0**		50	35.74	6.86E-18
kroB100	5000	**0**		50	57.1	6.86E-18	gr202	10100	**0**		50	63.26	6.86E-18
kroC100	5000	12	2866	50	65.74	6.86E-18	ts225	11250	**0**		50	136.72	6.86E-18
kroD100	5000	**0**		50	66.94	6.86E-18	gr229	11450	**0**		50	143.68	6.86E-18
kroE100	5000	**0**		50	60.4	6.86E-18	gil262	13100	**0**		50	16.98	6.86E-18
rd100	5000	**0**		50	39.88	6.86E-18	a280	14000	**0**		50	19.8	6.86E-18
eil101	5050	1	4093	50	7.64	6.86E-18	lin318	15900	**0**		50	69.88	6.86E-18
lin105	5250	20	3209.9	50	50.9	6.86E-18	rd400	20000	**0**		50	41.54	6.86E-18
pr107	5350	**0**		50	85.18	6.86E-18	fl417	20850	**0**		50	38.26	6.86E-18
pr124	6200	50	521.26	50	95.34	1.15E-14	gr431	21550	**0**		50	152.64	6.86E-18
bier127	6350	**0**		50	141	6.86E-18	pcb442	22100	**0**		50	70.88	6.86E-18
ch130	6500	1	1624	50	33.3	6.86E-18	rat575	28750	**0**		28	20.54	6.86E-18
gr137	6850	**0**		50	91.02	6.86E-18							

At first sight of table 3, the difference between the two scenarios seems to be broad. ILS cycles prematurely when comparing solutions with their fitness values, with only a few visited local optima. On the other hand, the hash function enables the algorithm to explore extensively, having to make a decision (restart, strong perturbation,...) when the cycle arises. With a $p-$value almost equal to zero, the Mann Whitney U Test confirms that the ILS algorithm always detects cycles earlier with the fitness function.

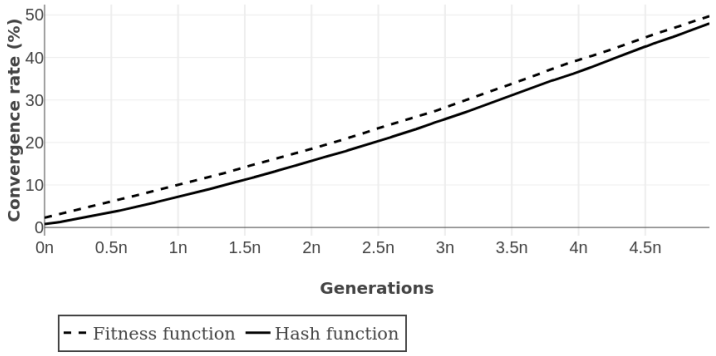
4.2 Convergence Speed

Population-based algorithms can also be exposed to misleading information provided by the fitness function. One of the stopping criteria in these metaheuristics is the convergence rate [12], i.e. the similarity of solutions within a population. We now analyse the convergence speed of two population-based algorithms.

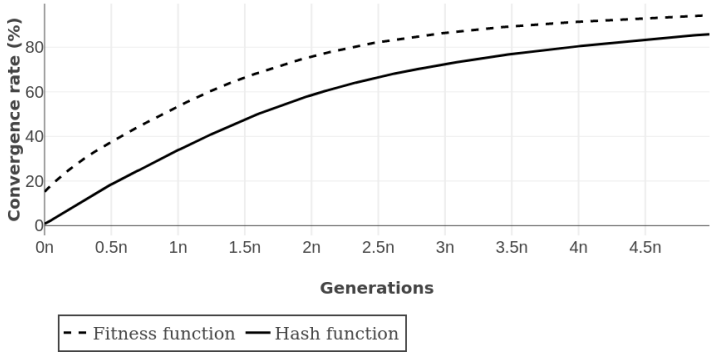
The first one is a GA [13] implemented with a tournament selection, one-point crossover and an elitist replacement strategy. The second one is a memetic

algorithm with the same genetic operators and the steepest descent applied to each new individual with 2-Opt.

For each scenario of f_{comp}, figure 2a (resp. 2b) displays the average (of the 39 studied instances) evolution of the convergence rate in the $5 \times n$ first generations for our genetic (resp. memetic) algorithm. The evolution is represented with dashed curves for $f_{comp} = f_{fit}$ and solid ones for $f_{comp} = \eta$.



(a) Genetic Algorithm



(b) Memetic

Fig. 2: Evolution of the convergence rate for population-based metaheuristics during $5 \times n$ generations

The two figures reveal the incorrect information given by the fitness function regarding the convergence rates. The difference is tight between the two cases of f_{comp} for the GA. In contrast, the memetic shows a wider gap between the two curves. This means population-based metaheuristics can run for more generations to explore new solutions and regions of the search space.

4.3 Applying hash function on metaheuristics

The previous results of this section warn us of the misguided analysis caused by the fitness function as a comparison function. A hash function can then be more effective in providing the algorithm with better results. Table 4 lists the results of three metaheuristics using η and f_{fit}, respectively, and implemented as follows: (i) An ILS with the steepest descent and perturbations with different strengths depending on the state of the search, run with $10 \times n$ iterations. (ii) A genetic, and (iii) memetic algorithms run with the same operators used earlier, with a stopping criterion of 90% on the convergence rate and a strictly equal number of evaluations for each variant of f_{comp}. The table shows the average gap from the optimal solution (formula 2) to the best solution found in each scenario of f_{comp}.

Table 4: Iterated Local Search, Genetic and Memetic Algorithms run with η, then f_{fit} as a differentiating function.

Instance	ILS		GA		Memetic		Instance	ILS		GA		Memetic	
	η	f_{fit}	η	f_{fit}	η	f_{fit}		η	f_{fit}	η	f_{fit}	η	f_{fit}
eil51	0.81	**0.58**	**187.76**	190.95	**0.53**	0.70	pr144	**0.01**	**0.01**	**920.35**	929.78	**0.00**	0.01
berlin52	**0.00**	**0.00**	193.64	**192.26**	**0.00**	**0.00**	ch150	1.88	**1.74**	**522.68**	527.60	**0.37**	0.47
st70	0.29	**0.21**	**296.54**	296.98	**0.09**	0.18	kroA150	1.56	**1.54**	**604.99**	612.51	**0.28**	0.33
eil76	**2.19**	2.23	**246.20**	253.87	**0.38**	0.85	kroB150	1.28	**1.26**	**601.41**	614.83	**0.19**	0.25
pr76	**0.09**	0.13	**291.06**	297.99	**0.04**	0.05	pr152	**0.21**	0.22	**942.39**	949.39	0.11	**0.10**
gr96	0.64	**0.58**	**389.74**	391.14	**0.22**	0.23	u159	**0.99**	1.02	**674.87**	681.97	0.05	**0.01**
rat99	**1.59**	1.76	**393.90**	399.59	**0.27**	0.34	rat195	**3.99**	4.51	**619.70**	622.17	**0.83**	1.09
kroA100	**0.25**	0.26	**473.64**	476.42	**0.04**	0.06	d198	**1.03**	1.07	**729.34**	732.31	**0.16**	0.18
kroB100	**0.56**	0.57	**442.16**	444.14	0.14	**0.11**	gr202	**2.34**	2.43	**421.79**	426.32	**0.29**	0.30
kroC100	**0.32**	**0.32**	**482.51**	490.31	**0.07**	0.08	ts225	**0.35**	**0.35**	**871.65**	873.22	**0.01**	0.03
kroD100	0.77	**0.70**	**444.51**	450.29	**0.15**	0.24	gr229	**2.26**	2.31	**658.88**	673.00	**0.34**	0.39
kroE100	**0.66**	0.75	**461.56**	464.54	0.21	**0.19**	gil262	**3.01**	3.22	**765.70**	775.37	**0.25**	0.64
rd100	0.65	**0.62**	**423.15**	424.12	**0.20**	0.25	a280	**2.94**	3.69	**890.03**	897.99	**0.26**	0.71
eil101	**2.91**	2.98	**303.01**	311.78	**0.48**	1.34	lin318	**2.53**	2.65	**979.80**	991.38	**0.36**	0.41
lin105	0.18	**0.13**	**508.88**	516.17	**0.02**	0.03	rd400	**3.91**	4.37	**983.71**	994.41	**0.57**	0.66
pr107	0.34	**0.29**	**761.09**	772.48	**0.09**	0.11	fl417	**0.88**	1.05	**2,897**	2,927	**0.22**	0.45
pr124	0.03	**0.01**	**745.45**	749.41	0.02	**0.01**	gr431	**3.02**	3.12	**965.36**	971.80	**0.54**	0.60
bier127	**0.86**	0.92	**313.17**	320.58	0.20	**0.19**	pcb442	**3.59**	3.99	**1,095**	1,102	**0.64**	0.71
ch130	1.31	**1.30**	**471.45**	478.66	**0.31**	0.40	rat575	**5.44**	6.52	**1,204**	1,205	**0.89**	1.92
gr137	1.18	**1.04**	**559.23**	564.66	**0.13**	0.16	**average**	**1.46**	1.55	**659.97**	666.57	**0.25**	0.38

The exposed results confirm the positive effect of using a hash function in different metaheuristic classes, especially for large instances. These improvements are achieved by non-biased runs of the above-mentioned algorithms. While the fitness function led to wrong cycles and premature convergence, the hash function allows the algorithm to make fairer differentiations and then make the right decisions at the right moments.

5 Discussion and conclusion

Using the fitness function to compare solutions can be harmful to many metaheuristics. This is due to the high number of collisions caused by the fitness function and the significant repetitions in its values. This paper proposes a new effective hash function with respect to the existing ones in the literature. The number of collisions caused by our function η is zero on almost all the generated samples and can be improved by comparing the pair (f_{fit}, η) of values. An analysis of different state-of-the-art heuristics unveiled the positive effect of using a hash function as a comparison tool. While the fitness function misleads the search process to short cycles, we observed longer explorations when using a hash function. A similar effect was noticed on population-based algorithms where the convergence rate increases more slowly with hash values. These improvements were reflected on their respective metaheuristics by reaching better solutions.

While this paper tackled the TSP as one of the most used COP, others can also take advantage of the proposed hash function or by designing new ones. We envisage then for our future works to explore new problems with different solution representations (permutations or binary strings). Genetic programming can be used to produce unbiased, and possibly problem-independent, hash functions.

References

1. Brownlee, A.E., Woodward, J.R., Swan, J.: Metaheuristic design pattern: surrogate fitness functions. In: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation. pp. 1261–1264 (2015)
2. Brucker, P.: Scheduling Algorithms. Springer Berlin Heidelberg (2007). https://doi.org/10.1007/978-3-540-69516-5, https://doi.org/10.1007/978-3-540-69516-5
3. Cai, S., Su, K., Sattar, A.: Local search with edge weighting and configuration checking heuristics for minimum vertex cover. Artificial Intelligence **175**(9-10), 1672–1696 (2011)
4. Croes, G.A.: A method for solving traveling-salesman problems. Operations research **6**(6), 791–812 (1958)
5. El Krari, M., Guibadj, R.N., Woodward, J., Robilliard, D.: Introducing a hash function for the travelling salesman problem for differentiating solutions. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. p. 123–124. GECCO '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3449726.3459580, https://doi.org/10.1145/3449726.3459580
6. Fay, M.P., Proschan, M.A.: Wilcoxon-mann-whitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules. Statistics surveys **4**, 1 (2010)
7. Flood, M.M.: The traveling-salesman problem. Operations research **4**(1), 61–75 (1956)
8. Hartung, E., Hoang, H.P., Mütze, T., Williams, A.: Combinatorial generation via permutation languages. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 1214–1225. SIAM (2020)

9. Hifi, M., Michrafy, M., Sbihi, A.: A reactive local search-based algorithm for the multiple-choice multi-dimensional knapsack problem. Computational Optimization and Applications **33**(2-3), 271–285 (2006)
10. Hoos, H.H., Stützle, T.: Stochastic local search: Foundations and applications. Elsevier (2004)
11. Koopmans, T.C., Beckmann, M.: Assignment problems and the location of economic activities. Econometrica **25**(1), 53–76 (1957)
12. Langdon, W.B., Poli, R.: Foundations of genetic programming. Springer Science & Business Media (2013)
13. Larrañaga, P., Kuijpers, C., Murga, R., Inza, I., Dizdarevic, S.: Genetic algorithms for the travelling salesman problem: A review of representations and operators. Artificial intelligence review: An international survey and tutorial journal **13**(2), 129–170 (4 1999)
14. Loiola, E.M., [de Abreu], N.M.M., Boaventura-Netto, P.O., Hahn, P., Querido, T.: A survey for the quadratic assignment problem. European Journal of Operational Research **176**(2), 657–690 (2007)
15. Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. The annals of mathematical statistics pp. 50–60 (1947)
16. Michalewicz, Z., Fogel, D.B.: How to solve it: modern heuristics. Springer Science & Business Media (2013)
17. Pierskalla, W.: The tri-substitution method for the three-dimensional assignment problem. Canadian operational research society journal **5**(2), 71 (1967)
18. Reinelt, G.: Tsplib—a traveling salesman problem library. ORSA journal on computing **3**(4), 376–384 (1991)
19. Túlio A.M., T., Thibaut, V., Tony, W.: Heuristics for vehicle routing problems: Sequence or set optimization? Computers & Operations Research **105**, 118 – 131 (2019)
20. Woodruff, D.L., Zemel, E.: Hashing vectors for tabu search. Annals of Operations Research **41**, 123–137 (1993)
21. Woodward, J.R., Bai, R.: Canonical representation genetic programming. In: Proceedings of the First ACM/SIGEVO Summit on Genetic and Evolutionary Computation. p. 585–592. GEC '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1543834.1543914, https://doi.org/10.1145/1543834.1543914