COMPLEXITY INTERNATIONAL

ISSN 1320-0682

Volume 02

April 1995

Preventing Diversity Loss in a Routing Genetic Algorithm with Hash Tagging

Simon Ronald
Program and Data Optimisation Group
National Key Centre for Social Applications of GIS
Adelaide, Australia
Email: `dna@gisca.adelaide.edu.au`

Abstract:

A new technique called hash tagging is presented, which helps preserve genetic diversity in routing-type genetic algorithms (GAs). For each new child generated in a population, a hash tag is calculated by applying a hash algorithm to each gene value in the genotype during a GA run. We do not allow a genotype to be added to a population if its hash tag clashes with an existing hash tag in use by another population genotype. This technique prevents duplicate population genotypes and enhances the performance of the GA in the later stages of evolution. We show that the technique introduces a powerful local-searching mechanism in the later parts of an evolution run. We present simulation results showing that hash tagging is effective for our chosen test problem (a 30-town Travelling Salesperson Problem).

The problem of diversity loss in GAs

Routing Genetic Algorithms (GAs) have successfully been used to solve a wide range of NP-hard routing type problems such as the job shop scheduling and the Travelling Salesperson Problem (TSP). A GA simulation typically begins with a population of N randomly generated genotypes. Each genotype encodes a single potential solution to the routing problem. A steady-state generation is defined as the process of creating one or two child solution genotypes using genetic operators such as crossover or mutation. In a steady state GA, a newly created child replaces the worst genotype of the population. This process of child production is repeated until a stopping criteria is met, which normally occurs when thousands of iterations have occurred. Convergence occurs when the genotypes in a GA have encoded solutions which have migrated to the top of a local or global peak in the solution space. The population has often converged when the stopping criteria has been met. At this stage in the GA run, the genotypes in the population are very similar if not identical to each other. Thus, the population has homogenised on this local or global peak solution. If the GA converges to a local, non-optimal peak, before the run is finished, then it is very difficult for the GA to move out to a more-optimal neighbourhood in the solution space. This process of diversity loss is often the cause of *premature convergence* which is the early convergence on an inferior local maximum.

A large number of existing techniques are used to maintain diversity in GAs. These include maintaining large population sizes, employing low reproductive or parent-selection pressures, applying mutation to the genotype, restarting the GA with new random genotypes, employing parallel populations (with occasional interchanging of fit chromosomes between populations), and niche-formation techniques. We compare hash tagging with these techniques in [3].

The hash tagging technique

We present a technique called *hash tagging*, which is an efficient way of preserving diversity. When a new genotype is produced, a hash tag is calculated. A hash tag is an integer identification tag that uniquely defines the constituent genotype. This tag is calculated and associated with the genotype until it is replaced by a new genotype at a later generation. The hash tag is evaluated by applying a hash algorithm to the list of gene values (integers) which define the genotype. This new tag must be unique in the population before the newly created genotype can be inserted into the population. After the hash tag is created, we can check for tag uniqueness in time O(1). This compares favourably with O(N) lengthy comparisons that would be required to determine if a new genotype exists in a population of **N** genotypes. We will show how hash tagging prevents duplicate genotypes from entering a population, hence preserving a measure of genetic diversity.

Calculating a hash tag

After a genotype **x** has been created with an operator, we calculate a hash tag (or hash code), which we shall call $h(x)$. We will use the following notation in our description:

Symbol	Meaning
N	Population Size
x	Genotype x
$h(x)$	Hash Tag for Genotype x
$x(a)$	Gene Value at Location a in Genotype x
l	The number of genes per genotype
c	Constant value - prime number
v, i, r	Variable Integer Values

Aho *et al.* in [1], describe experiments which compare a variety of hash algorithms. They discuss a number of techniques that ignore parts of the hash string in order to reduce the time taken to produce a hash code. They also present a hash algorithm that they called *hashpjw*, which creates a hash code based on every value in the hash string. They showed that this particular hash algorithm best distributed nine different sets of test data (hash strings) throughout the hash code data range. In computing a hash tag for a genotype, we wish to consider all genes in the genotype. If we were to consider only, say, every fourth gene value in the genotype (skip over three out of four gene values), we would increase the chance that a number of slightly different genotypes would map to the same hash tag. This is a situation we wish to avoid. The hash algorithm must produce a different hash code even if two adjacent gene values are reversed in order. Such minor genetic variations are commonplace when a GA population begins to converge. Our GA hash tag algorithm must differentiate between such minor variations in the hash string. Therefore, we used a hash algorithm similar to *hashpjw* to compute a hash tag for a genotype. A hash tag is calculated with the following algorithm (HC):

	Algorithm - HC - Calculate Hash Value of Genotype x
HC1)	Choose a value of c to be a prime number $\geq 10N$
HC2)	$i \leftarrow 1, \quad r \leftarrow 0$
HC3)	$r \leftarrow (rl + x(i)) \ mod \ c$
HC4)	$i \leftarrow i + 1$
HC5)	if $i \leq l$ goto HC2
HC6)	output r

We first choose the hash range **c** in Step 1 of the HC algorithm to be a large prime number. The algorithm then loops through each gene values in the input genotype in Steps 2 through to 5. In Step 3 a running hash code is calculated using a multiplicative and modulo formula. The algorithm outputs the hash code **r** which lies in the range $0, \cdots, c-1$.

Preventing duplicate population members

To prevent duplicate population genotypes, we employ the following algorithm HT. The symbol e_a is defined as a 1-dimensional vector, where $e_a \in \{USED, FREE\}$ and $a \in \{1, 2, \cdots, c\}$. The symbol x_i represents genotype **x**, which is the **i**th genotype of the GA population.

	Algorithm HT. Genetic Algorithm with hash tagging
HT1)	Generate Initial Population of Genotypes x_i for $i \in \{1, 2, \cdots, N\}$
HT2)	$e_i \leftarrow FREE$ for $i \in \{1, 2, \cdots, c\}$
HT3)	$e_{h(x_i)} \leftarrow USED$ for $i \in \{1, 2, \cdots, N\}$
HT4)	Choose a genetic operator and select one or two parent genotypes
HT5)	Use selected operator to create a temporary child genotype y
HT6)	Calculate Hash Value $h(y)$ for child y
HT7)	if $e_{h(y)} = USED$ then goto Step HT4
HT8)	$e_{h(y)} \leftarrow USED$
HT9)	Select a genotype in the population which will be replaced (x_w) - i.e. the population-worst genotype
HT10)	$e_{h(x_w)} \leftarrow FREE$ Reset old hash flag.
HT11)	Replace Genotype x_w with new child. $x_w \leftarrow y$
HT12)	If the stopping condition is satisfied, exit, otherwise goto Step HT4

Why duplicates cannot be created in a population incorporating hash tagging

In a regular GA, duplicates enter a population in one of three ways: 1) duplicate genotypes appearing in the initial (randomly generated) population. 2) duplicate genotypes appearing when a newly created child is identical to one of its parents. 3) duplicate genotypes appearing when an operator creates a new child which is the same as an existing population genotype.

The hash tagging algorithm HT prevents scenarios 2) and 3) from occurring. It would not be difficult to modify the algorithm to preclude the first scenario from occurring as well. A combinatorial analysis shows that the probability that randomly-generated duplicate genotypes will appear in the initial population is very small.

Why hash overflow tables are not used

The standard hash algorithm maps a hash string to a hash code. Hash algorithms generate hash codes which contain less information than the hash string (that is, we create an integer hash code from a genotype of **l** integers). Therefore it is possible that many different hash strings may map to the same hash code. The standard technique of determining whether a given hash string **k** exists in a hash table, is to use a linked list of duplicate strings attached to each hash code. Firstly, we would determine the hash

code for **k**, which is $h(k)$; then we would determine if there are other strings $\{k_1, \cdots, k_s\}$ which share this hash code by referring to an overflow table associated with the hash code of **k**. We would then compare our hash string **k** with each of these other hash strings which share the same hash code. From this, if our hash string **k** is different from $\{k_1, \cdots, k_s\}$, we would conclude that our string **k** is unique.

However, we do not use this standard technique of overflow tables. When we calculate a hash code $h(x)$ from a GA genotype **x**, if $h(x)$ is already in use by another genotype in the population **y**, we conclude that **x** is probably equal to **y** and simply bar genotype **x** from entering the population. However, a cost of barring a genotype from the population is incurred as we must use a genetic operator to create a new alternative genotype. This recalculation of a genotype would be wasted effort if in fact $x \neq y$. Thus, there is a trade-off between the cost of implementing a full hash table (with the maintenance and lookup costs incurred with overflow lists) and the cost of regenerating genotypes when in the previously noted situation when $x \neq y$. To keep the number of these redundant recalculations down, we chose a large experimental value of **c = 65599** (which is also a prime number).

Why hash tagging improves local search capabilities of a GA

We shall see in Section 3 that hash tagging has a significant beneficial effect in the way genetic diversity is maintained in a GA run. We shall see that this can lead to desirable properties - that is, fast convergence to the global maximum. We shall argue why hash tagging works so well.

When hash tagging is employed, a GA begins to converge on a maximum **m** in the problem space, and no duplicate population genotypes are permitted. This restriction results in the genotypes encoding solutions which crowd around the maximum **m**. These solutions will each contain the highly fit schema which characterise the maximum **m**. At this stage of evolution, if a genotype **x** is produced which does not contain enough of the fit schemata required to compete with the other genotypes, then genotype **x** will become the population-worst genotype and will be exterminated in the next steady state generation. Thus, the genotypes crowding around **m** will differ mainly in schemata of low fitness. Since the GA finds it hardest to choose between low fitness schemata, hash tagging provides a shelter where the GA may take its time to choose between the lower fitness schemata. In this way, at the end of the GA run, we greatly reduce the situation where the GA incorrectly eliminates a marginally better schemata because of stochastic sampling.

Improving the hash tagging algorithm for cyclic chromosomes

It was observed through experimentation with the TSP problem, when hash tagging was employed, that in the last phases of the GA run, different genotype isomorphisms were produced. For example, because in the TSP, the last town is considered adjacent to the first town, the following two genotypes $x_1 = [1\ 3\ 2\ 5\ 0\ 4\ 6\ 7]$ and $x_2 = [5\ 0\ 4\ 6\ 7\ 1\ 3\ 2]$ encode two isomorphisms of the TSP circuit $(0,\ 4,\ 6,\ 7,\ 1,\ 3,\ 2,\ 5)$. In this example, the hash code of x_1 would most likely be different from the hash code of x_2. However, the two different genotypes encode the same point in the solution space. The operator used to created new children, the Edge Recombination Operator [4] considers both x_1 and

x_2 to be identical - as the same edge relationships are present in both genotypes, so the edge recombination operator is not advantaged by the presence of both genotypes. The second operator used - inversion - is also not advantaged by the differences between both genotypes, as inversion was designed to work on a cyclic genotype. Since neither operator is advantaged by the differences between both x_1 and x_2, the presence of both in the population constitutes a loss of diversity. However, this argument only applies for genotypes which are cyclic. The argument would not apply, for instance, if we were searching for Hamiltonian paths. For the cyclic TSP case, to improve the performance of the hash tagging technique, we removed most redundant isomorphisms from the population. This was achieved by ensuring that all genotype children encoded the gene value 0 in the first gene location; that is, $x_3 = [0,\ 4,\ 6,\ 7,\ 1,\ 3,\ 2,\ 5]$.

Simulation results

Experiment conditions

The GA used in all experiments was a steady state GA. In every case, the initial population was created by shuffling lists of town identifiers (gene values) in each genotype to create **N** random Hamiltonian circuits.

All of the graphs presented in this section represent a simulation curve (a run) obtained by taking the average over 10 different GA sub-runs. Each GA sub-run was seeded with a unique random number seed which guaranteed a different random number stream for each sub-run.

All of the following runs employed two operators.

1) The Edge Recombination Operator (ERO) [4]. This operator takes two parents and generates two children from the combined edge map of both parents.

2) The inversion operator. This operator takes a single parent and generates a single child. The operator works by copying the gene values from the parent to the child. The operator then randomly selects a length value α and a random starting gene position τ in the genotype. The operator then reverses the ordering of gene values in the chunk of α gene values starting at location τ. The operator will wrap the reversed chunk of gene values over the end of the genotype, if necessary. For example, for the gene $[0\ 2\ 5\ 3\ 2\ 4\ 1]$, if α is selected as 3, and τ is selected as 6 then the resulting child will be $[4\ 2\ 5\ 3\ 2\ 0\ 1]$. The value α was chosen with a linear graded probability from the values $2, \cdots, l/2$ with the value of 2 being 1.75 times more likely to be chosen than the value $l/2$. The value τ was chosen

uniformly at random from the values 1 through to **l**.

Throughout all experiments, the ERO was applied with a fixed probability of 0.4, and inversion was applied with a probability of 0.6. The operators were not applied together in the same steady state generation.

In each experiment, if the population size was greater than 1000, then a linear selection bias of 1.9 was used. This means that the population genotypes were ranked in order of increasing fitness, and the fittest population genotype was chosen on average 1.9 times more often the the worst population genotype. If the population size was less than or equal to 200, then a much gentler selection bias of 1.01 was used. This low selection pressure was found to be beneficial to all experimental runs.

The problem used was a 30-town TSP. The reader is referred to [4]) for a list of the co-ordinates for this problem. The fitness of a genotype was defined as the length of the tour resulting from that genotype, divided by the length of the known optimal tour length (420 units).

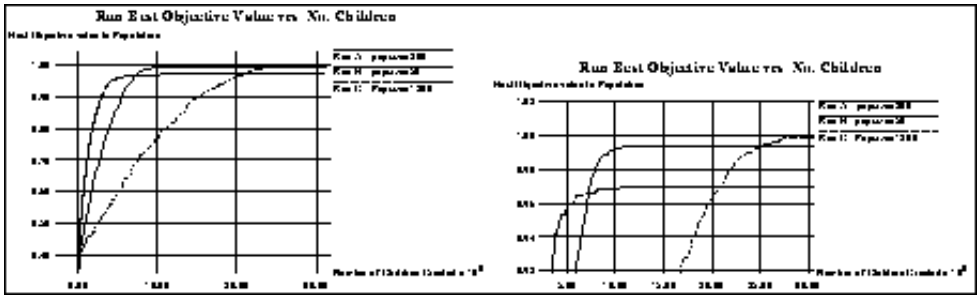

Figure 1: *(left)* 30-Town TSP without hash encoding. Run A uses a population size of 200. Run B population size of 50. Run C population size of 1200. *(right)* The same as left with region of convergence magnified.

We ran three experiments on a regular GA where we did not employ hash tagging. Three population sizes of 50, 200, and 1200 genotypes were used. We can see from Figure 1, that the larger population size (run C) converges to the best final solution fitness (0.9993). However, due to the lesser degree of genetic mixing, it takes 27,000 generations before this point is reached. The smaller population size of 200 (run A) converges to a smaller average final fitness value (0.993) in around 10,000 generations, but has obviously prematurely converged. The smallest population size of 50 (run B) genotypes also prematurely-converges in 7000 generations, and falls about 3% short of the global optimum (0.97). Experiments with population sizes less than 50 genotypes showed a marked increase in this percentage shortfall. The results in Figure 1 illustrate the results of genetic loss.

Results: Hash tagging used

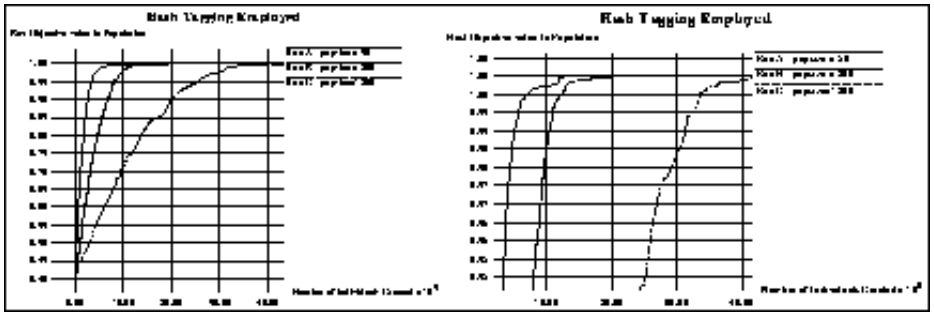

Figure 2: *(left)* Hash encoding employed. Run A uses a population size of 50. Run B uses a population

size of 200. Run C uses a population size = 1200. *(right)* The same as left with the region of convergence magnified.

Figure 2 shows the effectiveness of hash tagging. In each experiment, the hash tagging GA converged to the global optimum solution in 10 out of 10 runs. In the case of the Run A (50 genotypes in population), the best population genotype was reached (in all 10 sub-runs) before 14000 children were produced. The graphs in Figure 2 illustrate the effectiveness of hash tagging across all population sizes. In particular, the small population size of 50 genotypes showed no premature convergence whatsoever with the global maxima being reached in 10 out of 10 runs. The algorithm incorporating hash tagging was found to be extremely effective, even when solving the 30-town TSP in a tiny population size of 20 genotypes, when an average final solution of 99.8% of the true global optimum was reached in 10 distinct GA sub-runs. These results compare well with the results in Figure 1.

Results: Comparing diversity loss

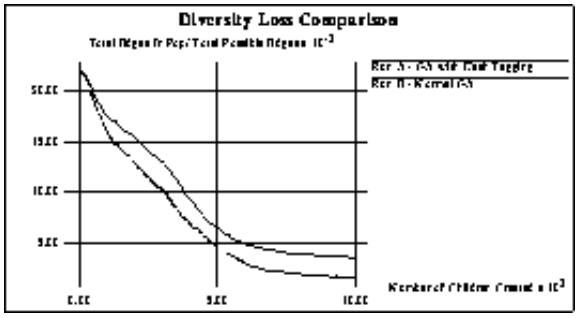

Figure 3: Fraction of the number edges present in population *versus* gen. 30-Town TSP Population size =200. Run A - Hash tagging (**c = 65599**). Run B - Standard GA.

Figure 3 shows the ratio of the total number of edges in the population divided by $N(N-1)$ (the total possible number of edges) *versus* the number of children produced. The GA employing hash tagging, maintained a steady state pool of 123 edges (at the end of the GA run). The regular GA population, however, finished up with a steady state pool of only 62 edges. Examination of the composition of the final population in run B, showed that this 62 edges was mainly due to two identical solutions encoded as two isomorphic genotypes where one genotype is a complete inverted ordering of the other. This suggests that we only have 2 extra edges over and above the single solution to which the GA population has converged. This low figure constitutes almost total diversity loss in the case of experiment B. What is most interesting, however, is the early stage in which hash tagging makes a noticeable difference in the population diversity. We can see from Figure 3, that only after 1000 new children, hash tagging begins to make a noticeable difference in the total number of edges remaining in the population.

Conclusions and future work

It is clear that hash tagging is effective for solving routing type problems such as the TSP in small population sizes. We have argued that the technique of hash tagging introduces a powerful local search mechanism in the latter part of a GA run. This local search has shown to be effective in solving small-sized TSP type problems without resorting to problem specific techniques such as 2-opt and 3-opt [2]. These results illustrate, that if diversity is preserved, then we do not need to resort to huge populations sizes or other less-effective techniques to find good solutions to our problem. Our results show, that in a

GA run, it is better to preserve a greater range of hard-earned solution schemata than to resort to large population sizes. Results in [3], show that hash tagging compares well with other techniques of diversity preservation for the same 30-town TSP test problem. These results suggest that hash tagging might be a viable alternative to current diversity preservation techniques. It is hoped that these results will also be useful in solving other types of problems than routing and scheduling problems.

Further investigation is required to determine if the technique performs as well for larger problem instances. Additionally, future investigation might uncover an efficient routine (of O(1)) that can determine whether a given genotype **x** has any matching population members in the near neighbourhood of **x** as defined by some distance metric such as a hamming distance (for binary encodings) or some scheduling type distance (for permutation encodings). If such a routine could be devised, then we could expect diversity in a GA run to be preserved at an even earlier stage than is possible with hash tagging.

References

1

Aho A. V., Sethi R. & Ullman J. D. (1986), *Compilers. Principles, Techniques, and Tools*, Addison-Wesley Publishing Company.

2

Taha H. A. (1987), *Operations Research*, 5th Edition, Macmillan Publishing Company.

3

Ronald S. P. (1994), "Preserving diversity in routing genetic algorithms: Comparisons with hash tagging", *Technical Report*, The University of South Australia, Department of Computer and Information Science.

4

Whitley D., Starkweather T. & Fuquay D. (1989), "Scheduling problems and traveling salesmen: The genetic edge recombination operator", in *Proceedings of the Third International Conference for Genetic Algorithms*, J. D. Schaffer (Ed.), Morgan Kaufmann, pp. 133-139.

About this document ...

Preventing Diversity Loss in a Routing Genetic Algorithm with Hash Tagging

The command line arguments were:
l2h `-dir test sr_hash.tex`.

The translation was initiated by Pam Milliken on Fri Aug 16 16:04:47 EST 1996

...isomorphisms
We do not, however, remove the isomorphisms in which a list of gene values is an inverted

ordering of the other.

...location
: The positional bias introduced in the genetic operators as a result of this shifting of gene values was removed.

...likely
: This value of 1.75 was obtained through brief experimentation with a range of values and was found to be a good choice. This is probably not the optimal value for this or other problems.
