Enhancing Genetic Algorithms by a Trie-Based Complete Solution Archive

Günther R. Raidl and Bin Hu

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Favoritenstraße 9–11/1861, 1040 Vienna, Austria
{raidl|hu}@ads.tuwien.ac.at

Abstract. Genetic algorithms (GAs) share a common weakness with most other metaheuristics: Candidate solutions are in general revisited multiple times, lowering diversity and wasting precious CPU time. We propose a complete solution archive based on a special binary trie structure for GAs with binary representations that efficiently stores all evaluated solutions during the heuristic search. Solutions that would later be revisited are detected and effectively transformed into similar yet unconsidered candidate solutions. The archive's relevant insert, find, and transform operations all run in time $O(l)$ where l is the length of the solution representation. From a theoretical point of view, the archive turns the GA into a complete algorithm with a clear termination condition and bounded run time. Computational results are presented for Royal Road functions and NK landscapes, indicating the practical advantages.

Key words: genetic algorithms, solution archive, revisits, tries

1 Introduction

Genetic algorithms (GAs) [7] are population-based metaheuristics for solving difficult optimization problems. This popular class of evolutionary algorithms often is able to find good approximate solutions within a huge search space in relatively short computation times. However, a drawback of GAs is that they usually do not keep track of the search history, and already evaluated solutions are often revisited. This in particular holds when the used selection pressure is rather high, the population size only moderate, or the variation operators do not introduce much innovation. In the extreme case, the population's diversity drops strongly and the GA gets stuck by creating almost only duplicates of a small set of leading candidate solutions, called *super-individuals*. In such a situation of premature convergence, it becomes very obvious that the heuristic search is not performing well anymore, and something must be changed in the GA's setup. However, also in scenarios which are believed to be rather well-configured, solutions are sometimes revisited and evaluated multiple times. Especially when the evaluation function is costly in terms of running time, such reconsidera-

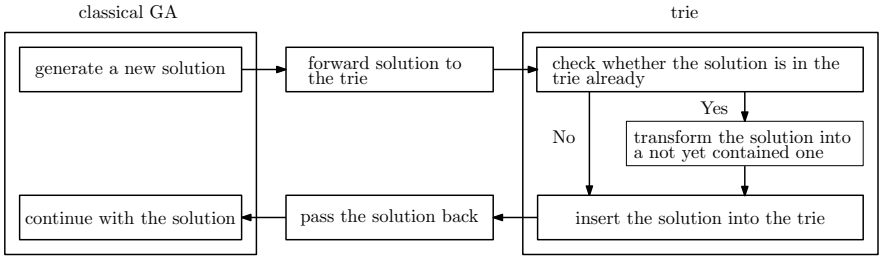

Fig. 1. Cooperation between GA and trie.

tions of the same candidate solutions may degrade performance substantially; re-evaluation in a general sense is a clear waste of precious computation time[1].

Various kinds of *population management strategies* have been suggested in literature to counteract premature convergence and to ensure a reasonable diversity in the population. They also reduce the number of revisits but are in general not able to completely avoid them. Rejecting new candidate solutions that are already contained in the current population of a steady-state GA often improves the situation and increases performance; it is therefore considered a good default strategy by many researchers. See e.g. [15] for a study in this respect. Nevertheless, revisits are obviously not entirely avoided in this way.

We consider a *complete solution archive* based on a memory-efficient *trie* data structure for GAs with binary solution representations in order to (a) efficiently detect already evaluated candidate solutions and to (b) efficiently transform them by typically small adaptations into yet unconsidered candidate solutions. Figure 1 illustrates how this archive is attached to the GA. In principle, this archive turns the GA into a *complete* optimization approach, which from a theoretical point of view is guaranteed to find an optimal solution in bounded time.

The computational overhead introduced by the archive is relatively low: The essential operations of inserting a new candidate solution, checking whether or not a new candidate solution is already contained in the archive, and transforming an already contained solution can all be performed in time $O(l)$, where l is the number of bits the binary solution representation has. Thus, the asymptotic time complexity of each iteration of the GA is not increased.

2 Related Work

Roland [17], for example, has shown that diversity loss through duplicates is a serious weakness in the steady-state GA model. He also proposed a simple way of removing duplicates by using a hash table to store all solutions currently in the population [16]. Going even further, Mauldin [12] compares newly generated solutions to all members of the population by their Hamming distance in order to select solutions for removal, hereby maintaining even higher diversity. Many

[1] with the exception of dynamic or stochastic scenarios where multiple evaluations might be intended to acquire updated or more reliable objective values.

similar approaches can be found in literature, however, they only take into account solutions of the current population and do not maintain a memory for the search history.

Looking more generally onto the field of heuristic search techniques, the popular *tabu search* (TS) [5] metaheuristic actually is based on the concept of maintaining a memory, usually called *tabu list*, that keeps track of the search progress in order to avoid cycling. Different kinds of memories are used, but typically only attributes of recently performed moves are recorded and used to forbid moves into parts of the search space. Usually, these tabu lists also have restricted length, and an appropriate choice of this length parameter often is a non-trivial task. Too long tabu lists may restrict the search too strongly, while too short lists will not effectively avoid cycles. The trend goes towards adaptive parameter control mechanisms, as e.g. in *reactive tabu search* [2]. Of course all these attribute-based TS approaches are not guaranteed to entirely avoid revisits.

Only very few TS approaches exist where entire solutions are directly or indirectly recorded in a memory and precisely those moves that would yield revisits are forbidden during the remaining search. The *reverse elimination method* [5] is one such example to realize what may be called *strict tabu search*. It is, however, relatively slow as at iteration n it requires a computation of order $O(n)$ to check whether or not a move is allowed. Therefore, Battiti and Tecchiolli [2] suggest to use classical *hashing methods*, see e.g. [1], or a *digital tree* [9] instead, by which the essential insertion and find operations can be performed in (expected) time $O(l)$, thus only depending on the size of the binary representation.

Battiti and Tecchiolli [2], however, further argue that such strict TS approaches might not work well in general as they often converge very slowly for problems where the local optima are surrounded by large *basins of attractions*, i.e., by large sets of candidate solutions that converge to the local optimum when performing local search. This slow convergence is related to a slow "basin filling" effect. Well-tuned attribute-based approaches in which larger parts of the search space are temporarily disallowed are therefore typically more effective. In addition, global optima might even become unreachable because of the creation of barriers consisting of already visited solutions. We believe that these negative aspects also require careful attention in the context of an archive-enhanced GA. However, the possible implications are obviously substantially less critical, as in contrast to TS a GA also includes other mechanisms for diverting the search and jumping over barriers (i.e., recombination and mutation).

Focusing again on evolutionary algorithms, solution archives have been used in general for several different purposes. Sometimes, elite solutions are explicitly stored in order to re-use them later in some way for improving search performance, see e.g. [4]. In particular in evolutionary algorithms for multiobjective optimization, explicit solution archives are frequently used for storing non-dominated pareto-optimal solutions [3]. The idea of *caching* objective values of visited solutions in order to avoid re-evaluations when they are revisited is quite natural in particular when the evaluation function is costly. For example Louis and Li [11] suggest to store solutions in a binary tree in such cases. Among

others, Kratica [10] and Ronald [16] described similar caching approaches using hash tables. Depending on the time effort of the objective function and the frequency of revisits, the total computation time can be lowered substantially. However, revisits are not prevented (or rejected) in these methods.

Aiming at completely avoiding revisits in a GA targeted towards continuous optimization problems, Yuen and Chow [19] proposed to use an archive based on a k-d tree data structure for storing all visited solution. When encountering a revisit, the corresponding solution is mutated in a special way in order to always derive a new, yet unvisited solution. This approach actually comes closest to the concept we pursue in the current work. Differences are, however, that we concentrate on a discrete binary search space and our trie-based approach is extended by certain features (e.g., randomization) in order to avoid a strong biasing when modifying already visited solutions.

More detailed, preliminary results of the current work can be found in the master thesis of Sramko [18], which has been supervised by the first author of the current work. In a follow-up master thesis by Zaubzer [20], the trie-based archive has been applied to a memetic algorithm for the multidimensional knapsack problem. Here, the knapsack constraints make the situation more complicated. Substantial problem-specific extensions have thus been considered for the archive in order to only produce candidate solutions on the boundary of the feasible region. Depending on the test instances and configuration, some benefits could be observed when using this archive. However, due to the strong repair and local improvement procedures embedded in this memetic algorithm, general advantages turned out to be rather small. Nevertheless, also this work gives a clear indication that the basic idea of enhancing a GA by a trie-based complete solution archive might be promising for many kinds of problems.

3 Trie-Based Solution Archive

We propose a solution archive for GAs based on a *binary trie*. This archive is, in principle, more generally applicable to metaheuristics for problems in which solutions are encoded as binary strings.

Tries are a class of data structures that are typically used to efficiently store a possibly very large set of strings [6]. Applications lie e.g. in language dictionaries for spell checking and translation or the indexing of documents for allowing a fast search. Different variants of tries exist, but they all have in common that the computational complexity of the essential insert and find operations only depends linearly on the string-length of the respective key, i.e., it is in $O(l)$. In comparison, balanced binary search trees would require $O(l \log n)$ time for these operations and typically also require significantly more memory. Hash tables are w.r.t. insert and find in the expected case similarly efficient as binary tries, i.e., they also only require $O(l)$ time, but they do not allow an efficient implementation of a *transform* operation that modifies an already stored solution into a similar new one. It shall also be remarked that *digital trees* [9] exhibit strong similarities to binary tries.

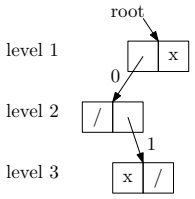

Fig. 2. Trie T containing solutions 010 and all with $x_1 = 1$.

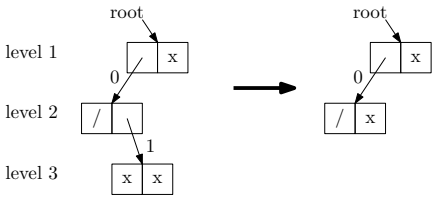

Fig. 3. Pruning the subtrie containing solutions 010 and 011.

3.1 Basic trie structure

The basic version of our binary trie is simple. It is a binary tree T of maximum height l. Each trie node t_i at level $i = 1, \ldots, l$ has identical structure: It consists of two entries $t_i.next[0]$ and $t_i.next[1]$ which are either pointers referring to successor nodes at the next level or are set to the flags *complete* or *empty*. The root node t_1 refers to the whole binary space $\{0,1\}^l$ and each other trie node t_i, $i > 1$, to a well-defined part of it: Considering binary vectors (i.e., candidate solutions) $x = (x_1, x_2, \ldots, x_l) \in \{0,1\}^l$, entries $t_i.next[0]$ and $t_i.next[1]$ of a node t_i always partition the corresponding space into the two subspaces containing those vectors with $x_i = 0$ and $x_i = 1$, respectively. Thus, the i-th bit of a vector x decides whether to go "left" or "right" at level i. An entry of *empty* means that none of the vectors lying in the corresponding subspace is yet contained in the trie, while an entry *complete* indicates the case that all corresponding vectors are contained (i.e., these solutions have already been visited by the GA). Figure 2 shows an exemplary trie containing the solution 010 and all solutions with $x_1 = 1$ ('x' denotes the *complete*-flag and '/' the *empty*-flag).

To check whether or not a solution x is stored in T, the search algorithm steps down the trie beginning at the root and follows $t_i.next[x_i]$ in each level i. If a *complete*-flag is encountered during this process, x is contained in T; if *empty* is encountered, x has not yet been inserted.

Adding new, not yet contained solutions works similarly, but new trie nodes must eventually be created when encountering an *empty* flag, and after considering the last bit a corresponding *complete*-flag is stored.

An important principle to keep the trie small is to prune a subtrie when all solutions corresponding to it have been added, i.e., if $t_i.next[0] = t_i.next[1] =$ *complete*, the respective pointer to t_i in the previous level is replaced by a *complete*-flag; see Fig. 3, where 011 has additionally been inserted. The special situation that the root pointer becomes *complete* thus indicates that all solutions of the whole search space have been added, and the GA can be terminated returning its best found solution as (proven) optimum.

3.2 Avoiding revisits by transforming solutions

One of the most important features of our solution archive is the ability to transform an already contained solution x, which would lead to a revisit in the

GA, into a typically similar but yet unconsidered solution x'. By "similar" we mean that the Hamming distance between x and x' is low. Considering the example of Fig. 2, if the GA's variation operators yield $x = 010$ again, this solution can be modified to $x' = 011$, which is then inserted in T leading to the situation shown in Fig. 3.

More generally, the basic idea of the transformation is to go back to some previous node at the search path of x whose alternate branch $p.next[1 - x_i] \neq$ *complete*, i.e., contains at least one yet unconsidered solution. Here, at this deviation position, we set $x_i = 1 - x_i$ and go down this alternate subtrie following the remaining bits of x whenever possible, i.e., unless we encounter a *complete*-flag in which case we choose again the alternate branch that must contain at least one unconsidered solution. We distinguish two algorithm variants w.r.t. the selection of the deviation position:

Deepest Position Transformation (DPT): In this basic variant the last (deepest possible) deviation position is always chosen. While this method probably is the most straight-forward one, it has the disadvantage of a strong biasing towards modifying bits at higher positions while keeping bits at lower positions unchanged. In fact, only when large portions of the search space have already been covered, bits at lower positions will be considered.

Random Position Transformation (RPT): To substantially reduce the above mentioned biasing, a random choice from all feasible deviation positions is made in the RPT algorithm variant. Otherwise, this method behaves in the same way as DPT.

A more detailed pseudo-code covering both transformation variants is given in Algorithm 1. We first search for x, then go back to the deviation point and go down once more in order to insert the transformed solution. The algorithm can be implemented in time $O(l)$.

3.3 Randomized trie structure

Using the basic trie structure where trie nodes on level i are always bounded to the bits on the i-th position has a significant weakness. Especially when using DPT for handling duplicates, we already observed that there is a strong bias towards some genes being changed much more frequently than others. In particular, bits at higher positions are always tried to be modified first. This bias typically results in repeated, rigid patterns when visualizing visited solutions in the search space. In general, when an intensive search around an incumbent solution has been performed and the trie is already moderately filled, the transform operation might need to flip not just the single bit at the deviation point but more bits in the course of finding a yet unconsidered solution, resulting in larger Hamming distances. As DPT considers the positions always in the strictly same order, this effect is significantly amplified, and transformed solutions with larger distances, i.e., less similar solutions, are created earlier and/or more frequently. This behavior can be compared with the well-known effect of *primary clustering* in hash tables when using linear probing as collision handling strategy.

Algorithm 1: Transform Solution

Input: solution $x = (x_1, \ldots, x_l)$; algorithm variant $var \in \{\text{DPT}, \text{RPT}\}$
$p = root$; $devpoints = ()$
// search for x storing possible deviation positions in $devpoints$
$i = 1$
while $i \leq l$ **and** $p \neq complete$ **do**
  if $p.next[1 - x_i] \neq complete$ **then**
    $devpoints = devpoints \cup (i, p)$
  $p = p.next[x_i]$
  $i = i + 1$
if $var = \text{DPT}$ **then**
  (i, p) = last entry of $devpoints$ // go back to last feasible deviation position
else
  (i, p) = random entry from $devpoints$ // go back to a random dev. position
$x_i = 1 - x_i$ // actually deviate by flipping bit i
while $i \leq l$ **do**
  if $p.next[x_i] == complete$ **then**
    $x_i = 1 - x_i$
  if $p.next[x_i] == empty$ **then**
    $p.next[x_i]$=new trie node $(empty, empty)$
  $q = p$
  $p = p.next[x_i]$
  $i = i + 1$
$q.next[x_i] = complete$ // insert transformed x in T
prune trie nodes of type $(complete, complete)$ bottom up
return x

The described RPT variant reduces these weaknesses substantially, but it is still not able to avoid a biasing entirely. As an alternative or additional improvement, we consider the randomization of the trie structure itself. The idea is to use individual, in general different bit orders on different search paths of the trie. Trie nodes at depth i are no longer related to the bit at position i, i.e., x_i, but a deterministic pseudo random (or hash) function is used to calculate the specific bit position j related to a given trie node t_i in dependence of the whole path from the root to t_i. Fig. 4 illustrates this randomized trie variant. One must be careful in the choice of the underlying pseudo random function. For example, classical Park-Miller random number generators are unsuitable as there are correlations between input and output data. We used the "pseudo data-encryption standard" algorithm *ran4* [14].

4 Computational Results

We present test results on two classes of standard benchmark problems from the binary domain: The Royal Road functions and NK fitness landscapes.

A more-or-less standard steady-state GA was used, which derives in each iteration one new solution by performing tournament selection with replacement,

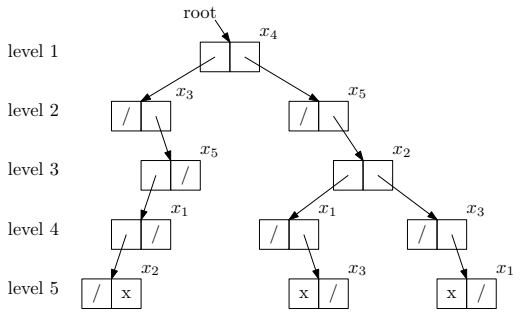

Fig. 4. Solutions 01100, 10011, 01111 in a trie with randomized structure. Nodes are labeled by their related bits of the solution vector.

Table 1. Considered GA variants.

algorithm	trie	transformation	trie structure
std	not used	–	–
tdb	used	deepest position	basic
trb	used	random position	basic
tdr	used	deepest position	randomized
trr	used	random position	randomized

always applying single point crossover, and mutating each bit with probability $1/l$. This new solution always replaces the population's worst solution. When the archive is attached, classical mutation is turned off and replaced by the transform operation of the archive in the case already visited solutions are obtained from crossover. Initial solutions were randomly created, the population size was 100, and the tournament selection group size 10. We compare the GA-variants and trie configurations summarized in Table 1.

In case the solution archive is not used (variant std), a classical duplicate elimination strategy as described in [16] is applied in the GA, i.e., a newly derived solution is only accepted in the population if it is different to all other solutions therein and discarded otherwise. The necessary duplicate-checks are efficiently performed by means of a hash table. Tests without any duplicate elimination and without the suggested archive yielded consistently clearly worse results, and we therefore do not include them here. All experiments were performed on a Pentium 4, 2.8 GHz PC with 1GB RAM.

4.1 Results on Royal Road functions

Royal Road functions were specifically designed for evaluating GAs with their crossover operators by Mitchell et al. [13]. They are defined for binary strings $x \in \{0,1\}^l$ on a set of hierarchically created schemas $S = \{s_1, s_2, \ldots, s_n\}$, and the objective is to maximize $\sum_{s \in S} o(s)\sigma_s(x)$, where $o(s)$ is the order of schema s (i.e., the number of defined bits) and $\sigma_s(x)$ is the binary indicator-function

Table 2. Results on Royal Road functions.

instance		std		tdb		trb		tdr		trr	
b	r	*avg*	*sd*	*avg*	*sd*	*avg*	*sd*	*avg*	*sd*	*avg*	*sd*
3	4	**36.00**	0.00	**36.00**	0.00	**36.00**	0.00	**36.00**	0.00	**36.00**	0.00
4	4	**48.00**	0.00	**48.00**	0.00	**48.00**	0.00	**48.00**	0.00	**48.00**	0.00
5	4	**60.00**	0.00	**60.00**	0.00	**60.00**	0.00	59.30	4.95	**60.00**	0.00
6	4	62.88	18.71	64.32	16.58	68.52	11.94	68.28	12.94	**68.64**	11.51
3	8	**96.00**	0.00	**96.00**	0.00	**96.00**	0.00	**96.00**	0.00	**96.00**	0.00
4	8	124.40	14.39	126.80	8.49	124.40	14.39	125.60	11.88	**128.00**	0.00
5	8	103.50	45.41	115.40	43.22	115.20	42.89	110.60	42.64	**125.30**	44.01
6	8	73.80	35.16	**92.64**	36.95	81.72	51.38	77.76	38.06	81.84	44.88
3	16	206.28	45.43	215.76	41.31	217.68	40.12	**226.98**	32.60	219.54	38.92
4	16	148.08	43.99	160.08	55.48	166.16	63.70	**168.00**	66.07	153.44	55.26
5	16	**106.50**	37.95	100.00	38.69	104.50	38.61	93.00	32.09	102.30	41.46
6	16	70.44	30.52	79.44	41.81	74.16	29.34	74.52	29.50	**82.68**	35.01

Table 3. Royal Road functions: Averages over all instances and Wilcoxon rank sum tests for each pair of GA-variants.

alg	*avg*	time	transforms	.vs std	.vs tdb	.vs trb	.vs tdr	.vs trr
std	92.96	0.02s	–	–	0.9953	0.9942	0.9677	0.9996
tdb	99.54	0.03s	312	0.0047	–	0.3854	0.2901	0.6849
trb	99.36	0.05s	297	0.0058	0.6150	–	0.3609	0.7099
tdr	98.67	0.03s	307	0.0323	0.7102	0.6394	–	0.9122
trr	100.15	0.05s	301	0.0004	0.3154	0.2904	0.0879	–

yielding 1 if x matches s and 0 otherwise. For details on S we refer to [13]; let b be the order of the smallest basic building blocks in S and $r = l/b$ be the number of them.

Table 2 lists results on differently parameterized Royal Road functions. All runs are terminated after 1000 iterations. Average final solution values (*avg*) and corresponding standard deviations (*sd*) of 100 independent runs are printed for each test case. Best values are marked bold.

Table 3 additionally lists the following total results over all considered Royal Road functions: Average objective values, average elapsed CPU-times when the best solutions were identified, the number of solution transformations (i.e., avoided revisits), and for each pair of GA variants the error probability of a one-sided Wilcoxon rank sum test for the assumption that one GA-variant performs better than the other.

We can observe that the GA without the trie archive performs worst in general. Among the different trie-variants, performance differences are rather small, but using the random deviation transformation and the randomized trie structure leads to noticeable improvements.

4.2 Results on NK landscapes

Introduced by Kauffman [8], NK landscapes serve as another popular benchmark suit for GAs with binary representations. The goal is to maximize the function

$$F(x) = \frac{1}{N} \sum_{i=1}^{N} f_i(x_{i_1}, x_{i_2}, \ldots, x_{i_K})$$

Table 4. Results on NK landscapes (random neighbors) over $N \in \{20, 50, 100, 300\}$, 10s CPU-time limit.

	std		tdb		trb		tdr		trr	
K	*avg*	*sd*	*avg*	*sd*	*avg*	*sd*	*avg*	*sd*	*avg*	*sd*
1	0.7090	0.0285	0.7089	0.0288	0.7086	0.0288	**0.7092**	0.0286	0.7089	0.0288
2	0.7351	0.0220	0.7354	0.0217	0.7364	0.0219	0.7361	0.0217	**0.7365**	0.0216
5	0.7603	0.0222	0.7623	0.0222	0.7611	0.0219	0.7609	0.0219	**0.7633**	0.0211
6	0.7628	0.0232	0.7631	0.0239	0.7641	0.0227	0.7645	0.0226	**0.7649**	0.0225
7	0.7595	0.0223	0.7583	0.0217	0.7592	0.0211	**0.7607**	0.0216	0.7600	0.0219
8	0.7567	0.0241	0.7583	0.0249	0.7582	0.0244	0.7597	0.0232	**0.7608**	0.0245
9	0.7545	0.0219	0.7526	0.0239	0.7557	0.0212	0.7560	0.0244	**0.7571**	0.0228
10	0.7505	0.0231	0.7522	0.0247	0.7507	0.0269	**0.7543**	0.0248	0.7528	0.0267

Table 5. NK landscapes (random neighbors): Averages over all instances and Wilcoxon rank sum tests for each pair of GA-variants, 10s absolute time limit.

alg	*avg*	*sd*	time	transforms	.vs std	.vs tdb	.vs trb	.vs tdr	.vs trr
std	0.7485	0.0290	3.47s	–	–	0.6851	0.9420	1.0000	1.0000
tdb	0.7489	0.0295	3.53s	52215	0.3149	–	0.8358	0.9999	1.0000
trb	0.7492	0.0294	3.69s	49602	0.0580	0.1643	–	0.9994	1.0000
tdr	0.7502	0.0294	3.48s	54003	0.0000	0.0001	0.0006	–	0.8472
trr	0.7506	0.0298	3.79s	50718	0.0000	0.0000	0.0000	0.1529	–

with $x \in \{0,1\}^N$. Each subfunction f_i takes gene x_i and K neighboring genes $x_{i_1}, x_{i_2}, \ldots, x_{i_K}$ into account and returns a value in $[0,1]$ according to a random value table. Hence there are N tables of size 2^{K+1} from which the values are read out. With increasing K, the coupling between particular genes rises and the problem becomes more complex. Two variants exist for choosing the neighboring genes $x_{i_1}, x_{i_2}, \ldots, x_{i_K}$: the *adjacent neighbors* version, where these genes are the closest ones to x_i, and the *random neighbors* version, where they are selected randomly distributed among all $x_1, \ldots, x_N$. We examine the NP-hard latter one.

Parameter N was set to 20, 50, 100 and 300 and K to 1, 2, 5, 6, 7, 8, 9 and 10. For each combination, we performed 50 independent runs and each run was terminated after 10s. Since the final solution values for different N but the same K are relatively similar, we decided to present here only accumulated results grouped by K due to space reasons. Table 4 shows these average final solution values and corresponding standard deviations for the standard GA and the four trie-enhanced variants. Though the average objective values here are close, the advantage of the trie is very obvious. In particular, the GA variant using the random deviation transformation together with the randomized trie structure was able to generate best average results most of the time. This becomes even more evident in Table 5 where average values over all K together with pairwise error probabilities of Wilcoxon rank sum tests are presented analogously to Table 3.

5 Conclusions and Future Work

In this paper we suggested the use of a complete solution archive based on a binary trie data structure for genetic algorithms using classical binary string

representations. The archive stores all solutions visited during the optimization process in a relatively compact way and provides the essential insert and find operations in time $O(l)$, i.e., independently of the number of already visited solutions. In contrast to classical objective value caching strategies, the archive further provides a new transform operation, which changes already visited candidate solutions effectively into in general similar but yet unvisited solutions. This procedure also runs in time $O(l)$. Randomized variants of this transformation procedure and the trie structure itself were proposed in order to minimize a biasing towards genes in certain positions being changed more frequently than others. From a theoretical point-of-view, a nice property is that the solution archive turns the GA into a complete optimization approach with a well-defined termination condition and bounded runtime.

Tests were performed on Royal Road functions and NK fitness landscapes. Although differences are not too large, we could observe that the use of the archive in general led to significantly better results and only moderate runtime increases. Especially the randomization of the transform operation and the trie structure also proved to be advantageous. More generally, we consider the proposed solution archive particularly promising for problems with expensive objective functions and relatively small search spaces, where solutions would otherwise be frequently revisited. There, the additional computational effort introduced by the archive becomes neglectable and the advantages can be expected to increase.

Further investigations and tests on more types of problems are necessary. In future work, we intend to study the application of trie-based solution archives in particular on hybrid GAs including e.g. local search or repair components for more complex optimization problems. This also includes problems where solutions are represented in other ways than binary strings. For them, the trie must be adapted appropriately. Constraints of a problem might further impose additional challenges on the trie and in particular its solution transform operator. For the multidimensional knapsack problem, some positive preliminary results have already been obtained [20]. Furthermore, we are currently working on archive-extended approaches for generalized network design problems where two dual representations are used at the same time. We also believe that it is promising to consider such trie-based solution archives for other metaheuristics besides GAs.

Last but not least, a particularly interesting aspect is the possibility to additionally calculate lower bounds (when considering a minimization problem) for individual subtries and to prune them (i.e., mark as completed) when this lower bound exceeds the objective value of the so far best solution. In this way, our approach becomes related to the well-known concept of branch-and-bound.

Acknowledgements

We thank Andrej Sramko, who helped in the implementation of the described concepts and did the testing as part of his master thesis [18]. This work is further supported by the Austrian Science Fund (FWF) under contract nr. P20342-N13.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1985.
2. R. Battiti and G. Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6:126–140, 1994.
3. K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms.* Wiley, 2008.
4. V. B. Gantovnik, C. M. Anderson-Cook, Z. Grdal, and L. T. Watson. A genetic algorithm with memory for mixed discrete-continuous design optimization. *Computers and Structures*, 81:2003–2009, 2003.
5. F. Glover. Tabu search - part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.
6. D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.
7. J. Holland. *Adaptation In Natural and Artificial Systems.* University of Michigan Press, 1975.
8. S. A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution.* Oxford University Press, 1993.
9. D. E. Knuth. *The Art of Computer Programming Vol. III: Sorting and Searching.* Addison-Wesley, 1973.
10. J. Kratica. Improving performances of the genetic algorithm by caching. *Computers and Artificial Intelligence*, 18(3):271–283, 1999.
11. S. J. Louis and G. Li. Combining robot control strategies using genetic algorithms with memory. In *Lecture Notes in Computer Science, Evolutionary Programming VI*, pages 431–442. Springer, 1997.
12. M. L. Mauldin. Maintaining diversity in genetic search. In *AAAI*, pages 247–250, 1984.
13. M. Mitchell, S. Forrest, and J. H. Holland. The royal road for genetic algorithms: Fitness landscapes and GA performance. In F. J. Varela and P. Bourgine, editors, *Towards a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, pages 245–254. MIT Press, 1992.
14. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 2nd edition, 1992.
15. G. R. Raidl and J. Gottlieb. On the importance of phenotypic duplicate elimination in decoder-based evolutionary algorithms. In S. Brave and A. S. Wu, editors, *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference*, pages 204–211, Orlando, FL, 1999.
16. S. Ronald. Complex systems: Mechanism of adaption. In R. Stonier and X. H. Yu, editors, *Complex Systems: Mechanism of Adaptation*, pages 133–140, Amsterdam, 1994. IOS Press.
17. S. Ronald. Duplicate genotypes in a genetic algorithm. In D. B. Fogel, H. P. Schwefel, T. Bck, and X. Yao, editors, *IEEE World Congress on Computational Intelligence (WCCI 98)*, pages 793–798, 1998.
18. A. Sramko. Enhancing a genetic algorithm by a complete solution archive based on a trie data structure. Master's thesis, Vienna University of Technology, Institute of Computer Graphic s and Algorithms, Vienna, Austria, February 2009.
19. S. Y. Yuen and C. K. Chow. A non-revisiting genetic algorithm. In *IEEE Congress on Evolutionary Computation (CEC 2007)*, pages 4583–4590. IEEE Press, 2007.
20. S. Zaubzer. A complete archive genetic algorithm for the multidimensional knapsack problem. Master's thesis, Vienna University of Technology, Institute of Computer Graphics and Algorithms, Vienna, Austria, May 2008.