Biol. Cybern. 73, 61–68 (1995)

Biological Cybernetics

Genetic algorithm with alphabet optimization

Gábor J. Tóth [1], **Szabolcs Kovács** [2], **András Lörincz** [1]

[1] Department of Photophysics, Institute of Isotopes, The Hungarian Academy of Sciences, H-1525 Budapest, P.O. Box 77, Hungary
[2] János Bolyai Institute of Mathematics, Attila József University of Szeged, H-6720 Szeged, Hungary

Received: 26 August 1994/Accepted in revised form: 13 January 1995

Abstract. In recent years the genetic algorithm (GA) was used successfully to solve many optimization problems. One of the most difficult questions of applying GA to a particular problem is that of coding. In this paper a scheme is derived to optimize one aspect of the coding in an automatic fashion. This is done by using a high cardinality alphabet and optimizing the meaning of the letters. The scheme is especially well suited in cases where a number of similar problems need to be solved. The use of the scheme is demonstrated with such a group of problems: the simplified problem of navigating a 'robot' in a 'room.' It is shown that for the sample problem family the proposed algorithm is superior to the canonical GA.

1 Introduction

The last decades have seen a revolution in heuristic programming. One of the underlying reason is the rise of the so-called adaptive approach. The traditional method of solving a real-life problem requires (1) building up a model, (2) using that model to find the optimal solution (with either algorithmic or heuristic approach), and (3) implementing the solution. This method is not always viable, as it involves steps that necessarily produce errors: both modeling and implementing are error-prone. The adaptive approach aims to overcome the problem posed by such errors by eliminating the model and using a heuristic program, testing the solutions proposed by the program on the real-life system itself. It thus involves the following steps: (1) choosing a possible solution randomly, (2) implementing the proposed solution, (3) measuring its success, (4) if the solution is not good enough then using the heuristic algorithm to propose another solution and continue from step (2).

The adaptive approach outlined above has the advantage that it is largely insensitive to systematic errors and noise both in implementing the proposed solutions and in measuring their success. Due to this fact, adaptive methods have been widely used recently, which in turn has brought to life many powerful heuristics, including the simulated annealing (SA) and the genetic algorithm (GA); see Press et al. 1992 and Beasley et al. (1993) and references therein. They also proved to be useful in many problems, which, due to their size or complexity, were intractable with traditional techniques. Yet, however powerful these techniques are, as new problems come into sight the need arises for inventing more powerful techniques or increasing the power of existing ones. In the present paper, an extension of GA is suggested. It is designed to solve families of problems and to give the solutions in such a way that information on the problems can be extracted from them.

GA is biologically inspired. It maintains a number of solutions, usually referred to as a 'population of individuals.' At the beginning, the individuals are initialized, usually randomly. Each individual is tested for its 'fitness' (the quality of its solution), and the population is subsequently updated (a new generation is formed) through a number of so-called genetic operators. Then the process continues with testing the new generation, until the performance is sufficiently good.

Usually three genetic operators are used to update the population: selection, crossover, and mutation. First, a mating pool is filled with individuals from the current generation: this is achieved by the selection operator, giving individuals of higher fitness more chance. Then some pairs of individuals in the pool are subject to crossover. Crossover combines the solutions of the two parent individuals to produce one or two (potentially better) offspring, to replace either or both parents, respectively. Finally, each individual is subject to mutation with a certain probability. Mutation means a random change in the individuals. After mutation, the individuals in the pool are used as the next generation.

The above 'definition' of the genetic operators is rather vague; every application has to define them precisely. In 'canonical' GA (CGA), an often used selection scheme is proportional selection. In that scheme, first the fitness of each individual is rescaled, so that the fitness of the best individual is at most σ times the average scaled fitness and no scaled fitness is negative, where σ is a constant characterizing the selection pressure. The mating

Corresponding to: A. Lörincz

pool is then filled by choosing individuals in proportion to the scaled fitness. The solutions of the problem are coded as bit-strings. A well set implementation of the crossover for bit-strings is the so-called one-point crossover: The two individuals to be crossed over are aligned, cut at the same point, and the resultant half-strings are exchanged. Finally, mutation is implemented by flipping each bit of the strings with given probability.

For certain implementations of the GA, it has already been shown that it can find the optimal solution of any problem with probability 1 (see, e.g., Rudolph 1994). In practical applications, however, it is not enough just to find the solution; it has to be found as fast as possible. This limitation posed by time constraints, by computer power, or by experimental possibilities can be eased by using a coding scheme and a set of genetic operators that suit the problem. Much effort has been made to find good coding schemes and implementations of the genetic operators, and to introduce new genetic operators for benchmark problems such as the traveling salesman problem (Goldberg and Lingle 1985; Oliver et al. 1987). Unfortunately, this may take some time. Moreover, it demands much insight into the problem. As such insight is not always available and also to save time, it would be useful to automate this process.

There are a number of choices one has to make to implement the optimization of the coding. Probably the most important is whether or not it is to be done simultaneously with the optimization of the basic problem. Algorithms running the two optimizations in parallel have already been proposed. They include the module acquisition (MA) of Angeline and Pollack or Koza's automatically defined functions (ADF) (see, e.g., Kenneth E. Kinnear 1994). In this paper we propose an algorithm that follows the other alternative: solves the basic problem multiple, in fact numerous, times and uses their results to improve the coding in a separate off-line process.

Although the simultaneous method is more straightforward, the off-line algorithm may have its own advantages. Most importantly, this method is suitable in situations when the basic problem is indeed a large set of related problems. In such cases, optimizing the coding on some members of the problem family, we may derive an encoding that can facilitate the successful solution of the others. In situations in which the goal is to solve one single problem, it may be possible to invent a set of training problems that can be used to optimize the coding. Undoubtedly, the design of training tasks needs some insight into the problem, but this kind of insight seems to be more readily available.

Another option one has when designing the algorithm is whether each individual should independently optimize the coding or all of the individuals should use the same coding. The ADF uses the former, the MA the latter method. The choice of the off-line method can take advantage of the latter method and rules out the per individual optimization.

A further important question is what exactly we mean by the optimization of the decoding. Clearly, many possibilities exist, but the three algorithms agree at this point. A basic alphabet is chosen to code the problem. Then a higher cardinality alphabet is set up containing partial solutions expressed in terms of this basic alphabet. The GA then works on the high cardinality alphabet, and its individuals are decoded by concatenating the partial solutions corresponding to the letters.

Even after all the above decisions are made, one still has a substantial freedom in specifying the algorithm. The ADF, for example, uses a fixed-size alphabet that contains a number of predefined letters (functions, as ADF was developed for genetic programming) and some functions that are being optimized (these are the 'automatically defined functions'). The MA, on the other hand, does not limit the size of the alphabet. It contains a number of predefined letters (functions); other letters (functions) are defined and removed on the fly. However, there is no explicit optimization of the letters. The algorithm described below, the genetic algorithm with alphabet optimization (GAAO), uses another method: the size of the alphabet is fixed, there are no predefined letters, and the letters undergo explicit optimization.

It may be worth noting that when solving a family of problems as outlined above, the optimization of the alphabet (AO) can be thought of as a means of cooperation between the GA procedures solving different members of the problem family: they share information by using the same alphabet. The use of some kind of cooperativity has been studied in many similar contexts, see, e.g., Kohonen 1984; Ritter et al. 1989; Tóth and Lörincz 1993.

In Sect. 2 we describe the algorithm in detail. For demonstration, it is tested on a simple robot navigation problem: a model robot has to learn to navigate in a room with obstacles; this problem is described in Sect. 3. The results are presented in Sect. 4. The closing section gives some further comments on the algorithm.

2 The algorithm

In this section, we describe in detail both the GA implementation and the algorithm optimizing the alphabet. The flow of the algorithm is the following. First, an alphabet is chosen randomly. The GA is run with that alphabet on a randomly chosen problem. The result of the run is then used to improve the alphabet. The process continues with testing the newly formed alphabet, until the performance is sufficiently good. The flowchart of the algorithm is shown in Fig. 1. In the following subsections, the GA and AO algorithms are described in detail.

2.1 The GA part of GAAO

The implementation of the GA is based on the CGA described in the Introduction. There are a number of aspects in which our GA implementation differs from the standard. First, to allow a useful optimization of the alphabet, the GA works on a relatively high cardinality alphabet. Second, as the decoding of the letters can change, it is useful to allow the length of the GA

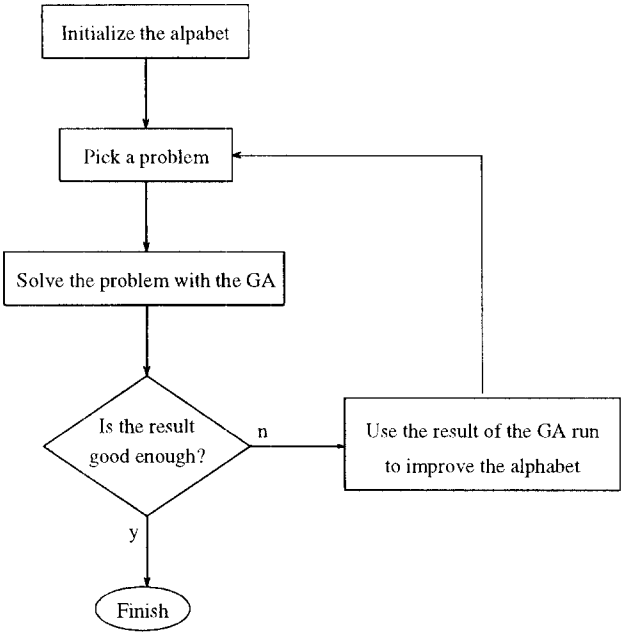

Fig. 1. The flowchart of the genetic algorithm with alphabet optimization (GAAO)

individuals to vary. To accommodate these features, the initialization, mutation, and crossover need to be modified.

The initialization is changed in two ways. As the length of individuals is not predefined, this procedure has to set the length of each individual. The length is chosen randomly from an exponential distribution of parameter λ_g over the set $\{1, 2, \ldots, l\}$: the probability that the length is k ($k = 1, \ldots, l$) is

$$p_k = \frac{e^{\lambda_g} - 1}{1 - e^{-\lambda_g l}} e^{-\lambda_g k} \tag{1}$$

The resulting individuals are filled with letters from the alphabet. The most straightforward way of doing so is to choose each letter with equal probability. This can be improved by associating each letter with a fitness value and choosing them in proportion to that fitness. Herein, this second method is used.

Operator crossover is based on the one-point crossover operator, but modified to accommodate the varying length of the individuals. When two individuals are to be crossed over, the length of the offspring needs first to be set. If k_1 and k_2 denote the length of the two parents, then the length k of the offspring is chosen randomly from the integers between k_1 and k_2, i.e., from the interval [min $\{k_1, k_2\}$, max$\{k_1, k_2\}$]. Then a cutting point c is chosen randomly from the integers of the interval [max $\{k - k_2, 0\}$, min$\{k, k_1\}$]. The first c letters of the offspring are copied from the beginning of the first parent, the remaining $k - c$ letters from the end of the second parent, i.e.,

$$g(i) = \begin{cases} g_1(i) & \text{if } 0 \leqslant i < c \\ g_2(i + k_2 - k) & \text{if } c \leqslant i < k \end{cases} \tag{2}$$

where $g(i)$, $g_1(i)$, and $g_2(i)$ are the ith bit of the offspring and the two parents. A graphical representation of the crossover is given in Fig. 2. This form of crossover assures that (1) the length of the individuals can change dynamically, (2) no individuals will be longer than l (there is no limitation on the length of the letters, see below), (3) crossing over two individuals of the same length will result in an individual of identical length, and (4) crossing over two identical individuals will result in an identical individual.

Mutation needs only a minor modification. In CGA, each bit to be changed is simply flipped. Here, each letter to be changed is replaced by another letter, which is chosen from the alphabet randomly, in proportion to its fitness.

2.2 The AO part of GAAO

The AO algorithm is also based on the GA. It contains a number of letters, collectively referred to as an alphabet (cf. individuals and population). The AO algorithm forms the letters from the units of the low cardinality basic alphabet. In the present example, the basic alphabet has two units: unit '0' and unit '1'. The high cardinality alphabet thus has the form of bit strings. Each letter is assigned a fitness that is based on the results of the GA runs. The alphabet is updated by genetic operators selection, crossover, and a new one: operator sticking.

The conceptual modification of the algorithm follows from the fact that the GA aims to find the optimal solution of a single problem, as opposed to this algorithm where the goal is to find a good set of different letters, while keeping their total number low to avoid an excessively high cardinality alphabet for the GA. While in GA it is in general perfectly permissible to have multiple copies of an individual, in AO we cannot afford this. Also, as the GA process may be given different problems each time, we cannot make sudden changes in the alphabet, otherwise a letter that is particularly useful in one problem but below average in others could disappear if a number of problems of the same type follow each other.

In each iteration, the fitness of the letters is changed according to the result of the GA run. Let us number the procedures according to the finished GA runs and let

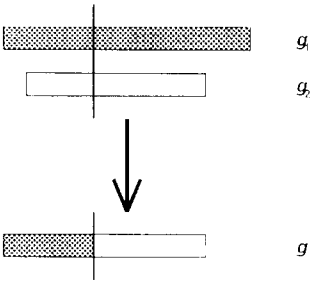

Fig. 2. Schematic representation of crossover. The *upper two boxes* represent the individuals to be crossed over, the *lower one* is the offspring. The *thick line* shows the cutting point

t denote this number; it will be called iteration count, or time. Let I_l denote the set of individuals using letter l at time t. The fitness of letter l is based on the fitness of the GA individuals using it:

$$f_l(t) = \alpha f_l(t-1) + (1-\alpha) \begin{cases} \max_{i \in I_l} f_i & \text{if } I_l \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

where $\emptyset$ denotes an empty set, f_l and f_i are the fitnesses of letter l and GA individual i, respectively, and α is a fading factor. In other words, the fitness of a letter at time t is the sum of the fitness of the best individual using it at time t weighted by $(1-\alpha)$ and the fitness of that letter at time $t-1$ weighted by α. The fitness value is used by the AO to update the alphabet, and also by the GA to select letters at initialization and mutation. As the GA needs the fitness of each letter, a newly formed letter will have to be given a fitness, too: it will be assigned the fitness of the worst unchanged letter.

The fading mechanism described above serves two purposes: by keeping the earlier results, it compensates the inherent deviation of the GA's performance, and also the uncertainty due to using different problems at the GA level. On the other hand, with the help of the fading mechanism, the adaptivity of the algorithm is maintained. However, this form of calculating fitness does not favor letters that are useful but seldom needed. If such letters are to be maintained, then (3) has to be modified.

The initialization of the alphabet is similar to the one used by our GA implementation. The length of each letter is chosen randomly from an exponential distribution over $[1, L]$ with parameter λ_a [see (1)]. The letters are filled with 0's and 1's randomly, so that no two letters are exactly the same.

Selection is achieved by deleting the worst $N_s + N_x$ letters (typically some 10%–20% of all individuals), where N_s and N_x are the number of letters produced by sticking and crossover, respectively. All the others survive in a single copy. The other two operators, i.e., sticking and selection, choose from the surviving letters independently of their fitness. This is to ensure that the relatively newly formed letters have the minimum bias against them; as the newly formed letters always have minimal fitness and also because of the fading mechanism, it will take some time while they get their true fitness value.

To replace the deleted letters, first the sticking operator is used. It chooses a GA individual randomly, in proportion to its fitness, and also a point in it. It then concatenates the content of the letter at that point with the content of the following letter of the same GA individual, thus forming a new bit string. If there is no letter containing the bit string formed, then it becomes a new letter. Otherwise, the content of the next letter of the same GA individual is concatenated to it. The process goes on until a unique letter is formed, or the end of the individual is reached. In the latter case, N_s is decreased for the current iteration, and N_x is increased instead. This procedure is chosen because in some cases, e.g., when all the individuals are the same and short, it may not be possible to form the prescribed number of new letters by sticking.

Sticking is followed by crossover, which is a variation of the one-point crossover. Two letters are chosen randomly from the surviving ones. Each of them is cut at a randomly and independently chosen point, and the resulting half-strings are exchanged. By choosing the cutting points independently, the property that crossing over two identical strings produces the same string is lost; however, there can be no two identical strings at any one time at all. Moreover, this form of crossover assures that the length of the letters can change dynamically. If the resultant letter already has a copy, then the newly formed one is discarded. The process continues until N_x new letters are constructed successfully.

It should be noted that this implementation of the AO lacks the operator mutation. The canonical mutation operator (flipping each bit with some probability) can be used, but at least in the application discussed here, it did not improve the result and was not used therefore. Also, symmetry might prompt for the counterpart of the sticking operator, i.e., an operator that cuts a letter into two and keeps only one-half. In some applications it may actually prove useful, but as crossover can both increase and decrease the length of the letters, it may not be needed.

3 The test problem

To illustrate the capabilities of GAAO, a simple problem family motivated by robotics was chosen. A 'robot' had to navigate in a 'room': it had to find the route between two randomly chosen points in the room (see Fig. 3). The robot had to avoid entering some parts of the room (the restricted zones), if it did, it was 'fined' for it; the initial and target points were always chosen outside the restricted zones. The robot had eight operators: four of them were interpreted as 'move one step up/down/left/right', the other four operators were meaningless, i.e., they did not move the robot. The problems corresponding to different initial and target points constituted the problem family. It should be noted that the choice of the problem is somewhat arbitrary; however, at this stage of

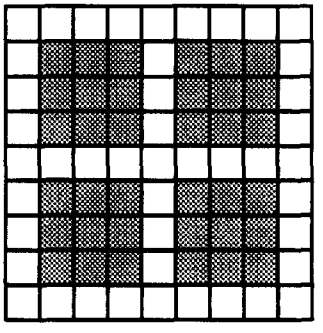

Fig. 3. The sketch of the 'room' the 'robot' was to navigate in. *Shaded regions* correspond to restricted zones. The operators up, down, left, right moved the robot one box in the corresponding direction

study the focus is on the algorithm, and the problem serves only as an illustration.

The motion of the robot was coded as a bit string. The string was interpreted in chunks of three bits, each possible triplet corresponding to one of the operators. The evaluation of the motion began with interpreting the first triplet. If it corresponded to one of the meaningful operators, the appropriate step was taken, unless it meant moving out of the room, in which case the robot did not move. If the new position was in a restricted zone or the robot had tried to move out of the room, it received a fine of F_b. If the chunk was meaningless, it received a fine of F_m. Then the evaluation of the motion continued with the next triplet, until the end of the string was reached. If the total number of bits was not a multiple of three, the last one or two bits were discarded. It may be worth noting that the four meaningless operators are not essential in this example problem; their inclusion serves only illustrative purposes: the letters of the alphabet will form a problem-specific hierarchy. There are different levels on this hierarchy, with differing low cardinality alphabets that may serve as starting points for the GAAO.

The quality q_s of solution s was calculated based on the assigned initial and target positions, the final position reached by the robot, and the sum of the fines the robot received during the motion:

$$q_s = \frac{(|x_i - x_t| + |y_i - y_t|) - (|x_t - x_f| + |y_t - y_f|) - F_l l_s - F}{(|x_i - x_t| + |y_i - y_t|)(1 - 3F_l)} \tag{4}$$

where (x_i, y_i), (x_t, y_t), and (x_f, y_f) are the initial, target, and final positions, respectively, l_s is the number of bits solution s contained, F_l is a constant, and F is the sum of all fines (F_b's and F_m's) received during executing the motion. This choice of the quality measure is somewhat arbitrary; other forms are possible as well. This form is suitable as it gives an approximately problem-independent (i.e., independent of the initial and target points) measure of quality in the sense that (1) the perfect solution of each problem corresponds to fitness 1.0 (except for problems where the initial and the target points are on the opposite sides of a restricted zone) and (2) no motion ($l_s = 0$) corresponds to zero fitness. Negative fitness values are possible because of the fines.

Computing the fitness of a GAAO solution involved a number of steps. First it was decoded from the high cardinality alphabet into binary. Two different procedures were tested. In concatenating mode, the bit strings corresponding to each letter of an individual were simply concatenated. In separating mode, however, if the number of bits of a letter was not a multiple of three, then the last one or two bits were discarded; only full triplets were concatenated. In either mode, the binary string was interpreted as explained above, i.e., in the order of consecutive chunks of three bits, to give the motion of the robot, and the corresponding quality measure q_s was used as the fitness of the GAAO solution. In the separating mode, l_s contained the number of cut-off bits, too.

The two decoding modes were introduced because either of them had its own advantages as well as disadvantages. The separating mode is better in the sense that it decreases the interdependence between letters: if a letter is meaningful in itself, it will be meaningful regardless of the preceding letters. In concatenating mode, this is not so: if the total number of preceding bits is not a multiple of three, then the meaning of the letter is completely changed. The concatenating mode was introduced to allow a meaningful comparison of the GAAO and the canonical (binary) GA. The CGA was emulated by switching off the AO and allowing only a binary alphabet to function. However, the binary alphabet and the separating mode cannot be used together: if a letter is shorter than three bits, it cannot possibly cause any motion. In all the other tests, however, the separating mode was used.

4 Results and discussion

The GAAO was used to solve the problem family described above. A number of tests were run to compare the GAAO with the CGA and to analyze the results. The parameters characterizing the GA and the problem are shown in Table 1 the parameters of AO are shown in Table 2. Each test was run 100 times to decrease the variance of the results. The figures always show the average of the 100 runs with its deviation.

First, the GA with binary alphabet was used to solve the problem, using the concatenating mode (see Sect. 3). To compensate for the shorter letters, the parameter of the exponential distribution used to generate the individuals (λ_g) was decreased to 0.08, which means that the average length of the individuals is approximately 12 bits (4 steps). The maximal length of individuals (l) was simultaneously increased to 50. It should be noted, however, that in practice l had little effect, both in this case

Table 1. The parameters characterizing the genetic algorithm (GA) and the problems. The same values were used in all tests, unless otherwise stated in the text

Cardinality of the alphabet	15
Number of individuals (n)	100
Number of individuals undergoing crossover (n_x)	90
Probability of mutation (p_m)	0.02
Maximum number of offspring of the best individual (σ)	2
Maximal length of individuals (l)	8
Parameter of the exponential distribution (λ_g)	0.5
'Fine' for length (F_l)	0.01
'Fine' for a meaningless code (F_m)	5.0
'Fine' for entering a restricted zone (F_b)	1.0

Table 2. Parameters of the optimization of the alphabet (AO) procedure

Maximal letter length (L)	10
Parameter of the exponential distribution (λ_a)	0.8
Number of letters replaced by sticking (N_s)	3
Number of letters replaced by crossover (N_x)	2
Fading rate (z)	0.75
Number of iterations	100

and in the others [see (1)]; it is used mainly for computational reasons.

The performance (fitness) of the binary algorithm is shown in Fig. 4. It is evident from the figure that the algorithm could not solve the problems successfully: on average, only one-third of a tour was made, or equivalent penalties were earned. This low average performance was caused by the long-distance tasks: if only a few steps are needed to reach the destination, the performance can be as high as 60%–70%, but it drops quickly with increasing distance (see Fig. 8). This can to some extent be explained by the low average length of the individuals. However, by increasing the average length (further decreasing λ_g), the search space is increased, and experience showed that this further lowered the overall performance.

It should not be surprising that the performance of the CGA is rather poor on this problem family, as these problems are difficult. In the step-by-step coding scheme, the value of the, say, tenth bit depends very much on the values of all the previous bits. If, for example, the starting point is the lower left corner, and the destination is the center, then either RRRRUUUU or UUUURRRR is a perfect solution. However, if the first three bits code an R, then only the first version is possible; anything else will have a much lower fitness, and thus there is a strong interdependency even between the first and the last letters. Having a good alphabet will not lessen the interdependence; however, it makes the search space smaller, and therefore the problem easier.

Figure 4 shows that the performance of the best of generation increases rapidly at the beginning, reaches a maximum and then decays to some degree. This unfavorable property is due to the high level of crossover (90% of the individuals undergo crossover). Newly formed fit individuals are easily disrupted before they can produce a sufficiently high number of copies. In theory, they can be reconstructed later, but because of the high interdependence, the diversity of the population decreases very rapidly, and so does the probability of reconstruction. By lowering the crossover rate, this decaying tendency can be eliminated. However, the AO part of the algorithm seemed to prefer this high crossover rate, and for consistency we kept this high value for all the runs. Alternatively, one could make the crossover rate depend on the iteration count number.

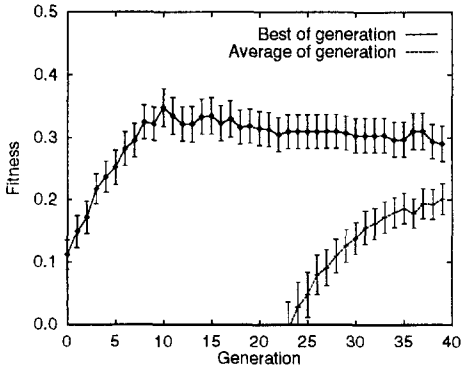

Fig. 4. The performance using a binary alphabet: average of 100 independent runs with its deviation

To demonstrate the value of a good alphabet, we tested the GA on a hand-optimized alphabet. In this, and in all of the following tests, separating decoding (see Sect. 3) was used. A 'perfect' alphabet was set up (see Table 3), and each letter was assigned the same fitness. The alphabet had letters coding forward steps in each direction of length 1, 2, and 4, and some further letters (coding turning) to make the cardinality 15. It should be noted that any motion can be made without turning letters, substituting, e.g., an RL with an R and an L.

The result of this run is shown in Fig. 5. Comparing it with the previous figure, the effect of a good alphabet is clearly pronounced: the efficiency of the search is increased. Historically, most of the GA implementations worked with binary alphabets. The theoretical reason for using binary alphabet is based on the so-called schema notion (see, e.g., Goldberg 1989). The schemata are similar to the GA individuals, except that they can contain a 'don't care' symbol in addition to the letters of the alphabet the GA is using. An individual is said to be an instance of a schema if in all positions either the schema and the individual contain the same letter or the schema contains a 'don't care' symbol. The GA can be thought of as processing a large number of schemata; this is called the implicit parallelism of GA (see, e.g., Goldberg 1989). The superiority of the binary encoding was explained by noting that it contains the highest number of schemata. Recently, several authors put forward different theories that no longer favored the binary encoding (see, e.g., Antonisse 1989). Our experimental results seem to align with the theories in favor of the higher cardinality alphabets.

The GAAO was now run in full strength, with the alphabet being optimized. The cardinality in this case was 30. After the GAAO run, the performance of the GA with the alphabet found by the GAAO was tested (see Fig. 6). For further comparison, the performance using only the better half of the alphabet (i.e., using again cardinality 15) was tested; the performance and the alphabet are shown in Fig. 7 and Table 4, respectively.

The alphabet of Table 4 deserves some discussion. The first thing to note is that the letters are fairly reasonable: there are no meaningless letters (although there were some in the worse half), and there is only one containing a cycle (RLRR). We have all the one and two-step letters (L, R, U, D and LL, RR, UU, DD); the remaining six code either three steps or turning. However, although we have all the one-step letters, their fitnesses differ widely: R is the fifth best, while U is only 15th. This is caused by the high fading rate: $\alpha = 0.75$. If, e.g., R were not needed in five consecutive tasks, its fitness

Table 3. The hand-optimized alphabet. Each letter was assigned the same fitness

L	LL	LLLL	RU
R	RR	RRRR	UR
U	UU	UUUU	LU
D	DD	DDDD	

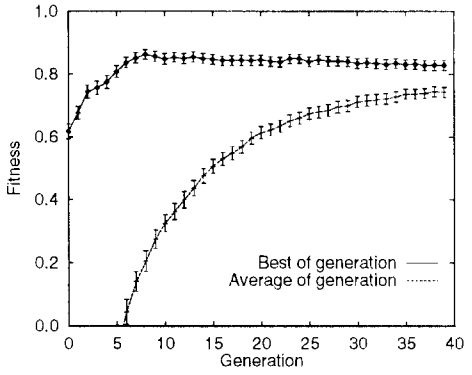

Fig. 5. The performance using a hand-optimized alphabet: average of 100 independent runs with its deviation

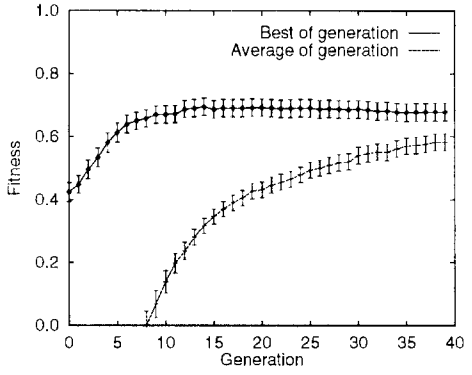

Fig. 6. The performance using the optimized alphabet: average of 100 independent runs with its deviation

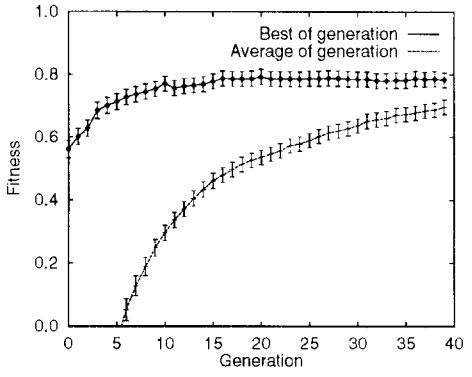

Fig. 7. The performance using the best half of the optimized alphabet: average of 100 independent runs with its deviation

Table 4. The alphabet obtained by optimization. The letters are ordered according to their fitness, from top to bottom and from left to right

Letter	Fitness	Letter	Fitness	Letter	Fitness
LL	0.36	UUL	0.20	D	0.10
LLL	0.33	UL	0.18	RLRR	0.09
UUUU	0.26	UR	0.15	DD	0.07
RR	0.24	UU	0.14	LD	0.06
R	0.24	L	0.13	U	0.05

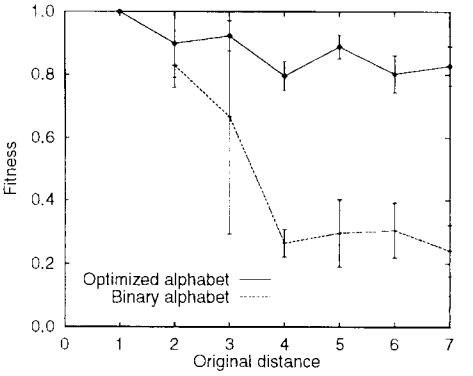

Fig. 8. The performance of the GA as a function of distance to be covered: average of 100 independent runs with its deviation

would be almost as low as U is now. By choosing a larger α value, this deviation can be decreased, at the cost of slowing down the optimization. Alternatively, α can be made time-dependent: starting from a higher value and decreasing during the optimization.

The results of using either the fully optimized alphabet or its best half are encouraging: (1) both outperform the binary alphabet, and the latter is comparable to the hand-optimized one, and (2) they are both superior to the hand-optimized alphabet in the sense that they needed less information about the problem being optimized. This knowledge was available in this instance: our simplified problem is fairly easy to understand. In more complex problems, such knowledge may not be available, and thus hand-optimization is not an alternative to GAAO, which can be used without such knowledge.

For a more detailed understanding of the difference between the CGA and GAAO, the quality of the best solution the GA has found with (1) binary alphabet and concatenating decoding and (2) optimized alphabet and separating decoding was plotted as a function of task distance (see Fig. 8). It is clear from the figure that the optimized alphabet is always better than the binary. The difference is most pronounced, however, for larger distances. This can partly be explained by noting that on average the first generation individuals are four steps long. However, with the current parameters, the breakdown of the binary GA occurs at around four steps, and the probability of producing an individual of at least 12-bit (four-step) length is almost 40%. So the difference is better explained by the fact that large distances correspond to very large search spaces, which severely limit the GA's performance. The longer letters of the optimized alphabet reduce the search space and make the problem tractable for the GA.

5 Further comments

In the previous sections we gave a detailed description of the GAAO algorithm proposed in the Introduction. It is designed to keep the most important features of GA: it is fully adaptive, in theory no information on the problem being optimized is needed (apart from the quality

measure of the solutions), and it is suitable for parallel implementation.

The GA's behavior is often described using the schema notion. The schemata are similar to the GA individuals, except that they can contain a 'don't care' symbol in addition to the letters of the alphabet the GA is using. The defining length of a schema is the largest distance between defined positions, i.e., positions containing anything but the 'don't care' symbol. The power of the GA is explained by its ability to find building blocks: useful, short defining length schemata. Unfortunately, strongly interdependent problems do not have building blocks in the above sense; because of the long range of the interdependence, the short defining length schemata are not meaningful in themselves.

The GAAO aims to overcome this limitation by optimizing the alphabet the GA can work on. The letters the AO is searching for can be viewed as generalized schemata that are continuous and position-independent, i.e., the defined positions are in a single block that can be placed to different positions of the solution string by adding 'don't care' symbols to the beginning and/or end of the string. The GA part of the algorithm can be seen both as (1) testing the alphabet and thus providing information to the AO to calculate the fitness of the letters, and (2) searching for schemata in the original sense but in terms of the letters, which can be used directly by the AO to build new letters with the sticking operator. As the GA is working on the alphabet provided by the AO, with improvement of the alphabet, the GA will be able to find more and more complex schemata, and thus the GAAO's ability to find useful schemata will be improved. If, however, there are no meaningful letters for the AO to find, the algorithm will revert itself to the CGA.

The AO can be seen as reducing the search space of the GA and decreasing the interdependence of the letters the GA is working on. The GAAO is most encouraging for large families of problems that need to be solved regularly and might possibly change in time. For such problem families, the development of the alphabet can be seen as a means of cooperation: the different problems of the family share information by using the same alphabet.

A practical field of application for GAAO, where meaningful letters for a family of problems can be expected, is robotics. The paper discussed a robotics-inspired problem to illustrate the algorithm: a simplified version of the robot navigation problem. The problem did indeed possess a number of meaningful letters. The GAAO could find a suitable set of such letters, and the algorithm proved to be significantly better than the CGA.

Another motivation of GAAO is the growing interest in building hybrid systems, i.e., systems combining adaptive and self-organizing methods with a backing knowledge-based system (KBS). One of the many efforts in building goal-oriented autonomous systems treats artificial neural networks (ANN) and KBS on an equal footing (Szepesvári and Lörincz 1994; Kalmár et al. 1994). This is achieved via building a directed graph from the ANN experiences and then using activation spreading (Thagard 1989; Collins and Loftus 1975) on the combined ANN graph and KBS tree to come to a decision. This decision-making procedure has several advantages: it is adaptive and fully parallel, and it offers the generation of subgoal hierarchy and thus can extend the KBS into the ANN (Kalmár et al. 1994). However, it suffers from the problem of combinatorial explosion: the memory requirement explodes when actions made of many steps are to be considered.

To ease the problem of combinatorial explosion, the model assumed that 'macroscopic actions' (e.g., step forward, turn left, etc.) are known. It has been shown in another work that the infinitesimal actions can be learned in a self-organizing fashion (Fomin et al. 1994). The self-organizing and action-planning procedures are rather similar in these works: the algorithms are built on very similar principles. The question then arises as to how one could fill the gap and develop macroscopic actions out of the infinitesimal ones. GAAO was developed with this problem in mind.

Acknowledgement. We would like to thank Csaba Szepesvári for helpful discussions.

References

Antonisse J (1989) A new interpretation of schema notation that overturns the binary encoding constraint. In: Shaffer JD (ed) Proceedings of the third international conference on genetic algorithms. Morgan Kaufmann, New York, pp 86–91

Beasley D, Bull DR, Martin RR (1993) An overview of genetic algorithms. Part 1. Fundamentals. Univ Comput 15:58–69

Collins A, Loftus E (1975) A spreading activation theory of semantic processing. Psychol Rev 82:407–428

Fomin T, Szepesvári Cs, Lörincz A (1994) Self-organizing neurocontrol. In: Proceedings of IEEE International Conference on Neural Networks, Vol 5, Orlando, Florida, pp 2777–2780

Goldberg DE (1989) Genetic algorithms in search, optimization and machine learning. Addison Wesley, Reading, Mass

Goldberg DE, Lingle R (1985) Alleles, loci, and the traveling salesman problem. In: Proceedings of an International Conference on Genetic Algorithms and Their Application, pp 154–159

Kalmár Zs, Szepesvári Cs, Lörincz A (1994) Generalization in an autonomous agent. In: Proceedings of IEEE International Conference on Neural Networks, Vol 3, Orlando, Florida, pp 1815–1817

Kenneth E, Kinnear J (1994) Alternatives in automatic function definition: a comparison of performance. In: Kenneth E, Kinnear J (eds) Advances in Genetic Programming, Chap 6. MIT Press, Cambridge, Mass., pp 119–141

Kohonen T (1984) Self-organization and associative memory. Springer, Berlin Heidelberg New York

Oliver IM, Smith DJ, Holland JRC (1987) A study of permutation crossover operators on the traveling salesman problem. In: Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms, pp 224–230

Press WH, Teukolsky SA, Vetterling WT, Flannery BP (1992) Numerical recipes in C: the art of scientific programming, 2nd edn. Cambridge University Press, Cambridge, UK

Ritter H, Martinetz T, Schulten K (1989) Topology conserving maps for learning visuomotor-coordination. Neural Networks 2:159–168

Rudolph G (1994) Convergence analysis of canonical genetic algorithm. IEEE Trans Neural Networks 5:96–101

Szepesvári Cs, Lörincz A (1994) Behavior of an adaptive self-organizing autonomous agent working with cues and competing concepts. Adapt Behav 2:131–160

Thagard P (1989) Explanatory coherence. Behav Brain Sci 12:435–467

Tóth JG, Lörincz A (1993) Genetic algorithm with migration on topology conserving maps. In: Gielen S, Kappen B (eds) Proceedings of the International Conference on Artificial Neural Networks. Springer, Berlin Heidelberg New York, pp 605–608