A Host-Parasite Genetic Algorithm for Asymmetric Tasks

Björn Olsson

Dept. of Computer Science, University of Skövde, Box 408, 541 28 Skövde, Sweden
Phone: +46-500-464716, Fax: +46-500-464725, Email: bjorne@ida.his.se

Abstract. We present a formalisation of host-parasite coevolution in Evolutionary Computation [2]. The aim is to gain a better understanding of host-parasite Genetic Algorithms (GAs) [3]. We discuss Rosin's [10] competetive theory of games, and show how it relates to host-parasite GAs. We then propose a new host-parasite optimisation algorithm based on this formalisation. The new algorithm takes into account the asymmetry of the two tasks: evolving hosts and evolving parasites. By self-adaptation the algorithm can find a suitable balance between the amount of resources spent on these two tasks. Our results show that this makes it possible to evolve optimal solutions by testing fewer candidates.

Keywords: Evolutionary Computation, Genetic Algorithms, Coevolution.

1 Introduction

A number of authors have investigated the use of coevolution as a method of improving current methods in Evolutionary Computation. Examples include cooperative coevolution [8] and competetive coevolution [9] - both of which have been used to design improved Genetic Algorithms (GAs). This paper focuses on host-parasite coevolution, which has previously been addressed in [4], [7], and [5]. A host-parasite GA uses two populations - a "host" and a "parasite" population - to represent candidate solutions and test cases. The central idea is to allow the test cases to coevolve with the candidate solutions, so that the algorithm self-adapts the set of test cases to be as challenging as possible at all stages of the optimisation process. Ideally, this results in a coevolutionary "arms-race" of continuous improvements in both populations.

The host-parasite relationship has obvious similarities with other coevolutionary relationships, such as predator-prey relationships, which have also been the subject of research in Evolutionary Computation [1]. In fact, host-parasite algorithms are often so similar to competetive algorithms, that no explicit distinction is usually made between them in the literature. The algorithms in [9], for example, share many of the properties of host-parasite algorithms, i.e. they use two separate populations where individuals from each population are fitness evaluated by being tested on members of the other population. However, most of the application examples in Rosin's work are tasks which are largely or completely symmetrical, i.e. the two populations are facing very similar tasks. In game playing, for example, the tasks for the two players are either identical or very similar. This means that the competetive GA is best designed so that both populations are evolved in a very similar manner.

We argue that a central distinction between competetive and host-parasite algorithms is that the latter are applied to tasks which are asymmetrical. In the

original work on host-parasite algorithms in [4], hosts evolved sorting networks whereas parasites evolved input sequences to be used as test cases. These two tasks are very different, and it seems natural to treat them as two distinct tasks. In this paper we will develop a host-parasite algorithm which explicitly takes the asymmetry of the two tasks into account, and we will show that this makes it possible for the algorithm to self-adapt the amount of effort spent on each.

2 Formalization of host-parasite coevolution

In this section we will develop a formalisation of host-parasite coevolution. This formalisation is based in part on the competetive theory of games in [10], but is is adapted and extended to be applied to host-parasite GAs. We will evolve a set H of host individuals, each representing a candidate solution to a problem. The population H will represent a subset of the possible candidate solutions $\mathcal{H}$. The exact size of $\mathcal{H}$ will depend on the problem (as well as on the chosen representation for candidate solutions), but for all problems of interest our population of hosts will just represent a small fraction of $\mathcal{H}$.

Simultaneously with evolving H, we will evolve a set P of parasites, which represent test cases. Again, P will represent a subset of the full set of possible test cases $\mathcal{P}$. In most cases, P will contain a small fraction of $\mathcal{P}$, but this may not be a necessary requirement for host-parasite algorithms to be advantageous. Our only restriction on the size of $\mathcal{P}$ is that it contains a finite number of test cases. We will use the test cases in P for fitness testing, so that the fitness of a host $h \in H$ is equal to the number of test cases in P that it solves. Conversely, the fitness of a parasite $p \in P$ will be equal to the number of candidate solutions in H that fails to solve p.

For any problem that we apply our approach to, we will assume that there is at least one perfect host individual in $\mathcal{H}$, i.e. a host which is able to solve every test case in $\mathcal{P}$. The goal for our optimisation algorithm will be to find such an individual by evolving the two populations H and P. It is important to realize that under the assumption that $\mathcal{H}$ contains a perfect host, the corresponding will not be true of $\mathcal{P}$, i.e. there will not be any test case which is not solvable by any host. Thus, the tasks of evolving H and P are done under different conditions, and this must be taken into consideration in the design of the algorithm. Unlike in many game playing tasks, the learning tasks we apply our algorithms to can be described as asymmetrical.

For formalisation purposes, we now introduce the following definitions, which will later be useful for describing and discussing our algorithms. We let $h \succ p$ denote the fact that the host h solves the test case p. Similarly, we use $H \succ p$ to denote that the set H of hosts solves the test case p, i.e. $\exists h((h \in H) \wedge (h \succ p))$. We will use $h \succ P$ to denote that h solves every test case in P, i.e. $\forall p((p \in P) \rightarrow (h \succ p))$. Given these definitions, the interpretation of $H \succ P$, will be $\exists h((h \in H) \wedge \forall p((p \in P) \rightarrow (h \succ p)))$.

Given this formal language, we observe that the fact that a perfect host exists in the space of possible hosts, can be denoted $\mathcal{H} \succ \mathcal{P}$. We also note that the goal for our optimisation algorithm is to evolve the populations until $H \succ \mathcal{P}$.

In order to understand the host-parasite GAs, we first describe an abstract optimisation process where a perfect host is found in a number of steps. This process starts with two empty sets of host and parasite individuals $H = P = \{\}$. We then alternately apply some search algorithm to H and P. In each step we add a host to H which solves all test cases in the current P, so that H contains a host that is perfect w.r.t. P. We then add a parasite to P, so that H no longer contains any host that is perfect w.r.t. P. This can be implemented by using the following algorithm:

let $t = 0$
let $H_t = P_t = \{\}$
repeat
  find a host h_t, such that $h_t \succ P_t$
  let $H_{t+1} = H_t \cup h_t$
  find a parasite p_t, such that $p_t \succ H_{t+1}$
  let $P_{t+1} = P_t \cup p_t$
  let $t = t + 1$
until $(H \succ \mathcal{P})$

Note that this algorithm is guaranteed to find a perfect host in a finite number of steps. For every iteration, we add a host that solves every test case found so far. We then add a new parasite, such that every host fails to solve at least one of the members of the new set of parasites. This new parasite must have found a flaw in the most recently added host. When, in the next iteration, we add another host which solves the full set of current test cases, this new host will necessarily be an improvement on the previously added host. In other words, this algorithm is guaranteed to make continuous improvements, and by a bootstrap process will reach $H \succ \mathcal{P}$ in a finite number of steps. The process forms a transitive chain of host and parasite pairs leading ultimately to an optimal host. The length of this chain can be referred to as l. We will later be concerned with the expected value of l, when discussing the time complexity of our algorithms. Of great interest, of course, is the relationship between l and the size of $\mathcal{H}$.

In order to develop host-parasite GAs, we need to consider in detail the method to be used in order to find h_t and p_t in each iteration. In our case, we use a GA to evolve these individuals. This raises a number of issues which will be addressed in the following sections.

3 An "asymmetric" host-parasite Genetic Algorithm

In order to guarantee that the algorithm will find $H \succ \mathcal{P}$, it is crucial that no individual is ever lost or deleted from H or P. More formally, the algorithm must ensure that

$$\forall h((h \in H_i) \wedge (j > i) \rightarrow (h \in H_j)) \tag{1}$$

and that the same condition holds for P. This is problematic since a GA does not guarantee that an individual which has been found in one generation will remain in all subsequent generations of the run. This is also true of the host-parasite algorithms studied in [4], [5] and [6]. To solve this problem, we will first design an algorithm where Condition 1 is guaranteed. We then relax this requirement in an algorithm where Condition 1 is likely to hold, but not guaranteed. In exchange for this uncertainty, we will improve the time complexity.

We now introduce a distinction between the evolvable populations H^e, P^e and the static populations H^s, P^s. Our GA will apply reproduction to evolvable populations, while static populations will serve only as fitness tests. More specifically, we will evolve H^e using the members of P^e and P^s as fitness cases, until $H^e \succ P^e \cup P^s$, i.e. until H^e contains at least one host h that solves all current test cases. We will add h to the current H^s, and then evolve P^e using H^e and H^s as fitness cases. Evolution of P^e will continue until every host in H^e and H^s fails to solve at least one of the test cases in P^e. We will then find the parasite $p \in P^e$ which h (i.e. the most recent addition to H^s) fails to solve, and add it to P^s. The process of alternately evolving H^e and P^e will continue until $H^e \succ \mathcal{P}$.

The following algorithm implements these ideas:

let $t = 0$
initialize random H_t^e and P_t^e
$H_t^s = P_t^s = \{\}$
while $(\neg(H_t^e \wedge \succ \mathcal{P}))$ do
  repeat
    evaluate and reproduce H_t^e using $P_t^e \cup P_t^s$ as fitness cases.
  until $(\exists h_t((h_t \in H_t^e) \wedge (h_t \succ (Pe_t \cup P_t^s))))$
  repeat
    evaluate and reproduce P_t^e using $H_t^e \cup H_t^s$ as fitness cases.
  until $(\neg((H_t^e \cup H_t^s) \succ P_t^s))$
  find p_t, such that $((p_t \in P_t^e) \wedge \neg(h_t \succ p_t))$
  let $H_{t+1}^s = H_t^s \cup h_t$
  let $P_{t+1}^s = P_t^s \cup p_t$
  let $H_{t+1}^e = H_t^e$
  let $P_{t+1}^e = P_t^e$
  let $t = t + 1$
done

For implementation purposes we must address the question of the population sizes of H^s and P^s. As noted earlier, the algorithm will form a transitive chain leading from the first host in H^s to the final perfect host. The sum of the population sizes of H^s and P^s in the final time step will be equal to l. Since H^s and P^s are used in fitness testing, it is crucial that we either reduce the expected value of l or the population sizes of H^s and P^s, while still ensuring that the algorithm will find a transitive chain to $h \succ \mathcal{H}$. Unfortunately, it is generally not possible to delete any individual from H^s or P^s without running a risk of losing the transitivity property. To see this, consider the extreme case of limiting population sizes of H^s and P^s to 1. In this case, every new member that we add to H^s replaces the previous member. It is easy to see that problems may arise given two hosts h_1, h_2 and two parasites p_1, p_2, such that $h_i \succ p_j$ **iff** $i = j$. The algorithm may alternate between h_1 and h_2 as members of H^s (while alternating between p_2 and p_1 as members of P^s). Similar examples can be found for greater population sizes.

In our implementation we treat H^s and P^s as queues where elements are added to the queue at one end and deleted at the other. We hypothesise that the element that has been in the queue for the longest time will be the one least crucial for maintaining the transitivity property.

4 Results

For ease of reference we will call our new algorithm AHPGA, for "Asymmetric Host-Parasite GA". AHPGA takes into account the asymmetry of the two tasks of evolving hosts and parasites, effectively treating these as two separate tasks. If one of the tasks proves more difficult than the other, AHPGA may be able to self-adapt the amount of effort spent on each task, so that a suitable balance is found. We will compare AHPGA with the Simple Host-Parasite GA, SHPGA, which was shown in [6] to give improved results over "standard" GAs.

In the first runs we used the 6 and 7-input sorting networks tasks, for ease of comparison with previous results in [6]. In table 1 and figure 1 we compare the results of AHPGA and SHPGA. For all runs, we used a population size of

30 for both the host and parasite populations. For AHPGA this means that the population sizes of H^e and P^e were fixed at 30 individuals. The static populations H^s and P^s were initially empty, and limited to contain a maximum of 30 individuals. In none of the runs was this upper limit of 30 reached, i.e. no individual was ever discarded from H^s or P^s. All runs were terminated when $3 * 10^5$ (6-input) or $3 * 10^6$ (7-input) individuals had been evaluated. It is important to realize that SHPGA evaluates and reproduces both populations in each generation, whereas one of the populations is static in AHPGA. This means that SHPGA evaluates up to twice as many individuals per generation as AHPGA.

	Converged runs		Evaluated solutions	
	6-input	7-input	6-input	7-input
AHPGA	49	50	$4.2 * 10^4$	$7.7 * 10^5$
SHPGA	48	3	$6.9 * 10^4$	$10.0 * 10^5$

Table 1. Statistics of likelihood and speed of convergence.

As table 1 shows, the average number of evaluated individuals before convergence in AHPGA is less than two thirds of the number in SHPGA. We also found that the fraction of generations during which the parasite population was evolving in AHPGA was only 0.02 for the 6-input task and 0.03 for the 7-input task. This illustrates in a striking way the asymmetru of the tasks - it is obviously very easy to evolve a test case p which the current best host is unable to solve, but very difficult to evolve a host which solves p, since 97-98% of all generations were used to evolve hosts. For the 6-input task it took on average less than 4 generations to evolve p, whereas it took on average 218 generations to evolve a host which solved p. For 7-input, the figures were 10 vs 535 generations.

By taking into account the asymmetry of the two tasks - evolving good solutions and evolving good test cases - AHPGA seems to find solutions by spending more time on the harder of the two tasks and less time on the easier task. Achieving such a balance is impossible in SHPGA, since it does not take the asymmetry of the tasks into account. Keeping in mind that AHPGA only spends 2-3% of all generations on evolving parasites, we realise that a very large proportion of the parasites generated in SHPGA are redundant.

5 Discussion and Conclusions

Our experimental results are quite preliminary since they are only from a single example application. The algorithm must be tested on a large number of examples before definitive conclusions can be made. It should also be kept in mind that there are both observed and potential problems with AHPGA in its current form. While the number of evaluated individuals before convergence is lower in AHPGA than in SHPGA, the number of applications of the fitness function is larger. While AHPGA evaluates 30 individuals per generation (with the population sizes we used) it uses at least as many calls of the fitness function as SHPGA (which evaluates 60 individuals per generations). The reason is that every call of the fitness function in SHPGA gives information on two individuals - one host and one parasite - whereas a fitness function call in AHPGA only gives information about the individual from the currently evolving population.

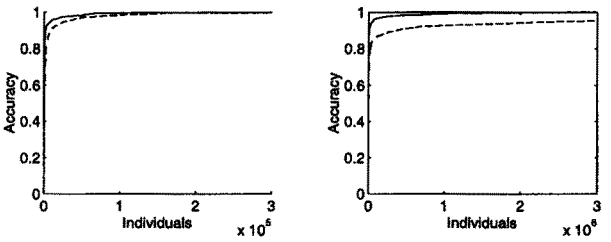

Fig. 1. Average accuracy for best host so far versus number of evaluated individuals for 50 runs on the 6 and 7-input sorting networks design task using AHPGA (solid line) and SHPGA (dashed line).

This work gives us more insight into the dynamics of a host-parasite optimisation process. It shows that the two populations face tasks that are very different, and may best be treated as two separate optimisation tasks. It exposes the often hidden assumption of symmetry behind competetive algorithms and shows that it may be unfortunate to apply this assumption when attacking tasks that are largely asymmetric. We may not yet have found the ideal way of taking asymmetry into account, but AHPGA shows that there are potential advantages of doing so. In other words, we see AHPGA as a starting point for improved host-parasite GAs, better suited to solve inherently asymmetric tasks.

References

1. D. Cliff and G.F. Miller. Co-evolution of pursuit and evasion ii: Simulation methods and results. In *From Animals to Animats 4: Proc. of the 4th Intern. Conf. on Simulation of Adaptive Behavior (SAB96)*, 1996.
2. D.B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence.* IEEE Press, 1995.
3. D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.
4. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In *Proceedings of the 2nd Conf. on Artificial Life*, 1992.
5. B. Olsson. Optimization using a host-parasite model with variable-size distributed populations. In *Proceedings of the 1996 IEEE 3rd International Conference on Evolutionary Computation.* IEEE Press, 1996.
6. B. Olsson. Evaluation of a simple host-parasite genetic algorithm. In *Proc. of the 7th Annual Conf. on Evolutionary Programming*, 1997.
7. J. Paredis. Steps towards co-evolutionary classification networks. In *Proc. the 4th Conference on Artifical Life*, 1994.
8. M.A. Potter. *The Design and Analysis of a Computational Model of Cooperative Coevolution.* PhD thesis, George Mason University, 1997.
9. C.D. Rosin. *Coevolutionary Search Among Adversaries.* PhD thesis, University of California, San Diego, 1997.
10. C.D. Rosin and R.K. Belew. A competetive approach to game learning. In *Proc. of the 9th Annual ACM Conf. on Computational Learning Theory*, 1996.