The Selfish Gene Algorithm: a new Evolutionary Optimization Strategy

Fulvio CORNO, Matteo SONZA REORDA, Giovanni SQUILLERO

Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy

http://www_cad.polito.it/

{corno, sonza, squillero}@polito.it

Keywords: Genetic Algorithm, Selfish Gene

Abstract. This paper proposes a new general approach for optimization algorithms in the Evolutionary Computation field. The approach is inspired by the Selfish Gene theory, an interpretation of the Darwinian theory given by the biologist Richard Dawkins, in which the basic element of evolution is the gene, rather than the individual. The paper defines the Selfish Gene Algorithm, which implements such a view of the evolution mechanism. We tested the approach by implementing a Selfish Gene Algorithm on a case study, and we found better results than those provided by a Genetic Algorithm on the same problem and with the same fitness function.

1. Introduction

The field of Evolutionary Computation is based on search and optimization algorithms that were inspired by the biological model of Natural Selection. Several different algorithmic paradigms, among which we find Genetic Algorithms, Genetic Programming, and Evolutionary Programming, were proposed after the Darwinian theory. Their underlying common assumption is the existence of a population of individuals that strive for survival and for reproduction. Under this assumption, the basic unit of evolution is the *individual*, and the goal of the algorithm is to find an individual of maximal fitness.

The work of the biologist R. Dawkins [Dawk89] has put evolution in a different perspective, where the fundamental unit of evolution is the *gene*, rather than the individual. This view is not in contrast with classical Darwinism, but provides an alternative interpretation key, that is formalized by the Selfish Gene Theory. In this theory, individual genes strive for their appearance in the genotype of individuals, whereby individuals themselves are nothing more than vehicles ("survival machines" in Dawkins' terminology) that allow genes to reproduce. In a population, the important aspect is not the fitness of various individuals, since they are mortal, and their good qualities will be lost with their death. Genes, on the other hand, are immortal, in the sense that a given fragment of chromosome can replicate itself to the offspring of an individual, and therefore it survives its death. Genes are selected by evolution on the basis of their ability to reproduce and spread in the population: the population itself can therefore be seen as a *pool of genes*. Due to the shuffling of genes that takes place during sexual reproduction, *good* genes are those that give higher reproduction probabilities to the individuals they contribute to build, when combined with the other genes.

The goal of this paper is to apply the shift of paradigm brought by the Selfish Gene Theory to the field of algorithmic optimization, and to develop a new approach in Evolutionary Algorithms, whose focus is on the fitness of genes, rather than of individuals. To give credit to Dawkins theory, we call this approach *Selfish Gene Algorithm* (SG). Starting from the Selfish Gene hypothesis, we develop a general framework in which to write optimization algorithms. Although the Selfish Gene theory is biologically equivalent to classical Darwinism, we expect SG to behave differently than Genetic Algorithms (GAs), due to a different focus of the optimization algorithm.

Following the Selfish Gene theory, the SG, as presented in this paper, neither relies on any crossover operator, nor needs to model a particular population. Instead, it works on a Virtual Population, which models the *gene pool* concept via statistical measures. Each potential solution is encoded as a genotype, where each *locus* can be occupied by one of several possible *alleles*. In the SG, different alleles fight to be present in a specific locus. The success of each allele is represented by its frequency in the Virtual Population and it is related to its *goodness*, but the frequency does not represent the fitness. Fitness calculation is performed at the phenotypic level, considering the full genome.

To verify the feasibility of the approach, we implemented a SG engine and we ran it on a test problem, the 0/1 Multiple Knapsack Problem. The comparison of the results with those

obtained by means of a Genetic Algorithm shows definite advantages for our approach.

This paper is organized as follows: Section 2 briefly summarizes the biological Selfish Gene theory, and Section 3 describes in detail the *Selfish Gene Algorithm*. A case study problem is described in Section 4, together with the relevant experimental results. Section 5 concludes the paper.

2. Summary of the Selfish Gene Theory

In 1976 English biologist Richard Dawkins wrote a book called *The Selfish Gene* [Dawk89], initially regarded as a work of radical extremism, followed in 1982 by *The Extended Phenotype* [Dawk82]. In these works he proposed a new theory for considering the Darwinian natural selection mechanism. Dawkins himself is a neo-Darwinist and he claims that his theory "is Darwin's theory, but expressed in a way that Darwin did not choose". He considers his own work "a logical outgrowth of orthodox neo-Darwinism, expressed as novel image".

Darwin's natural selection is based on the concept of *survival of the fittest*. The usual point of view is to consider the *individual* as the entity that can be more or less *fit to survive*. For instance, an individual can have a longer neck and, for this reason, can have more chances to survive in the world. This point of view leads to interpretation problems when trying to define exactly *who* or *what* is surviving.

If we follow Dawkins' claims [Dawk89], obviously, the individual itself does not survive. An individual only lasts for a while when we are looking at the whole evolution process. We can claim that the individual does not survive, but the genome of the individual is able to replicate itself into subsequent generations. Anyway, we should remember that all individuals are unique and sexual reproduction is not replication. Children are, in most cases, only half of one parent, grandchildren are only a quarter, and so on. A few generations later, the most an individual can hope for is a large number of descendants, each of whom bears only a tiny portion of him. Individuals are not stable things: they are fleeting. Genomes too are shuffled into oblivion, like hands of cards soon after they are dealt.

Rather than focusing on the individual organism, Dawkins proposes to focus on the genes, or, to be more precise, on little portions of the chromosome that he decided to call *genes*. Using the metaphor of the cards, we can say that genomes are jumbled up, but the cards themselves survive the shuffling. The cards are the genes, and the genes are not destroyed by any crossover, they merely change partners and march on.

The *survival of the fittest* is a battle fought by genes, not individuals. Only genes can be more or less suited to survive, because only genes can *survive* in the evolution. Individuals are simply vehicles, made up from the *blind* cooperation of different genes. If a gene is able to produce a useful characteristic, for instance a longer neck, individuals with such gene in their genome will have more opportunity to have offspring, thus such a gene will have a higher probability to be spread in the world.

The most interesting fact is that gene cooperation is *blind*. A gene is blind: it is not conscious, nor has it any idea about the genome it is part of. A gene can be a *good* gene or a *bad* gene, depending on other genes. For instance the gene causing the longer neck can be useful for a giraffe and potentially deadly for other animals, but the gene does not care about the animal it is in. Natural selection cares. Natural selection combines genes into fit genomes, without the need of any *global* information about individuals. Relations between genes caused, for instance, by pleiotropy and polygeny are *implicitly* considered by natural selection.

3. The Selfish Gene Algorithm

Dawkins' theory can be reflected from biology to computer science, and it leads to interesting applications in the field of Evolutionary Computation. Our goal is not to discuss the *Selfish Gene Theory* as a biology theory, but to use the concepts elaborated by Dawkins for developing a new kind of optimization algorithm.

3.1. Algorithm Definition

3.1.1. The Virtual Population

Traditional *Genetic Algorithms* rely on the concept of *population*. A population is a set of individuals; each of them has associated a fitness value, which measures the *goodness* of the individual. Time is divided into discrete steps, called *generations*. At each generation some new individuals are generated through crossover operators and some are discarded. The choice of which individuals are used for performing reproduction usually depends on their fitness. Usually, a mechanism called *elitism* is used to preserve best individuals through generations, giving them a sort of unnatural longevity, or even immortality.

For the Selfish Gene theory individuals are not so important, and the population is seen as just a store of genetic material. In this view, the *Selfish Gene Algorithm* does not consider and explicit population, and does not enumerate the individuals belonging to it. Rather, it uses an abstract model, called *Virtual Population* (VP), where the number of individuals, and their specific identity, are not of interest, and therefore are not specified nor stored. The VP aims at modeling the *gene pool* concept defined by Dawkins. For implementation purposes, we resort to a statistical approximation of the VP, by modeling and evolving some of its statistical parameters, described in what follows.

The evolution of the VP proceeds by an unspecified kind of sexual reproduction of its individuals. Since individuals are not explicitly listed, but implicitly represented, the SG models reproduction through its effects on the statistical parameters that model the VP.

As with other Evolutionary Algorithms, in SG an individual is represented by its genome. To avoid confusion, for each gene we will explicitly distinguish between its location in the genome (the *locus*) and the value appearing at that locus (the *allele*). Let g be the number of loci into the ge-

nome; each locus L_i (i=1...g) into the genome can be occupied by n_i different gene values, called alleles. The alleles that can occupy locus L_i are denoted with a_{ij} (j=1...n_i) and are collectively represented as a vector $\mathbf{A}_i = (a_{i1}, a_{i2}, \ldots, a_{in_i})$.

In the VP, due to the number of possible combinations, genomes tend to be unique, but some alleles might be more frequent than others. In the SG, the success of an allele is measured by the frequency with which it appears in the VP. Let p_{ij} be the marginal probability for a_{ij}, which conceptually represents the statistical frequency of the allele a_{ij} in locus L_i within the whole VP, regardless of the alleles found in other loci. Marginal probabilities of alleles in $\mathbf{A}_i$ for locus L_i are collected in the vector $\mathbf{P}_i = (p_{i1}, p_{i2}, \ldots, p_{in_i})$. The VP can therefore be statistically characterized by all marginal probability vectors $\mathbf{P} = (\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_g)$. Please note that $\mathbf{P}$ is not a matrix, because the number of alleles for each locus n_i can be different.

```
genome SG()
{
  genome B, G1, G2 ;
  initialize all pij to 1/ni ;
  B = select_individual() ; /* best */
  do {
    G1 = select_individual() ;
    G2 = select_individual() ;
    /* tournament */
    if (fitness(G1) > fitness(G2))
    {
      reward_alleles(G1) ;
      penalize_alleles(G2) ;
      if (fitness(G1) > fitness(B))
        B = G1 ; /* update best */
    } else {
      reward_alleles(G2) ;
      penalyze_alleles(G1) ;
      if (fitness(G2) > fitness(B))
        B = G2 ;
    }
  } while(steady_state()==FALSE) ;
  return B ;
}
```

Figure 1: *the Selfish Gene Algorithm*

Although a better statistical characterization of the VP is possible, for instance by taking into account joint probabilities, for the purpose of this paper a first-order approximation suffices.

3.1.2. Evolution Mechanism

The SG does not rely on a crossover operator; in fact, reproduction is performed implicitly and there is no crossover operator at all. In the SG (whose pseudo-code is shown in Figure 1) the VP evolves through a mechanism called *tournament*. Because there is no explicit definition of individuals in the VP, an individual is generated only when needed for competing in a tournament, and it is discarded immediately after. Two individuals are randomly selected from the VP, according to the allele frequencies in $\mathbf{P}$, and they are collated using a competition. In the competition, the fitness function for the two individuals is evaluated at the phenotypic level, and the one with higher fitness is considered to be the winner.

We assume that this trial occurs many times in the population, and after each competition the winner has the opportunity to reproduce itself, while the loser can not generate any offspring. The result is that all alleles belonging to the winner slightly increase their frequency in their respective loci in the VP at the expense of those belonging to the loser. The effect of sexual reproduction is implicitly modeled by the fact that the rewarded genes will be selected together with other alleles, in other loci, different from the ones belonging to the former winner: genes are shuffled and are combined in many different ways. In order to form a winning individual, the best value of each allele depends on the selection probabilities of other alleles in the genome, thus building a form of blind cooperation between genes. With this mechanism, alleles of the winner increase their selection probability, forming a positive feedback that drives a fast algorithmic convergence.

```
genome select_individual()
{
  genome H ;
  for (each locus i = 1..g)
    if ( random_number(0, 1) < pm )
      /* mutation */
      H[i] = random_allele(1, ni);
    else
      /* use probability Pi */
      H[i] = select_allele(Pi);
  return H ;
}
```

Figure 2: *Individual Selection*

Individual selection is further detailed in Figure 2, where an individual H is built by choosing which allele a_{ij} to put in each locus L_i using the probability reported in $\mathbf{P}$. To introduce further variability, a *mutation* can occur with a probability p_m, in which case the mutated allele is chosen in a completely random way.

```
reward_genes(genome H)
{
  for (each locus i = 1..g)
    pi,H[i] = pi,H[i] + εi ;
}

penalize_genes(genome H)
{
  for (each locus i = 1..g)
    pi,H[i] = pi,H[i] - εi ;
}
```

Figure 3: *Virtual Population Update*

The update of the probabilities p_{ij} according to the outcome of the tournament is performed by procedures `reward_genes` and `penalyze_genes`, detailed in Fig-

ure 3, where a constant ε_i is added/subtracted[1] to the relevant marginal probabilities for locus L_i. The value of ε_i determines the entity of the positive feedback, and therefore the balance between a fast convergence towards a local optimum and a broader exploration of the search space. Usually, all ε_i are set to the same value ε.

The algorithm is iterated until some stopping condition is reached. The function `steady_state` tests whether the VP evolution has reached a steady state, the *steady state* is defined as a state in which, for each locus L_i, there exists an allele a_{ij} whose probability is over a given threshold p_t (usual p_t values are around 0.95).

$$\forall i : \max_j(a_{ij}) > p_t \qquad (1)$$

When the condition (1) is met, all individuals modeled by the VP are very similar and the VP is not likely to evolve any more (individuals extracted with *select_individual* function can never be identical, because the mutation probability always preserves a little variability).

3.2. The SG in Optimization Problems

At the beginning of the evolution process, all alleles being equally probable, the SG makes the VP randomly drift, until an allele slightly increases its marginal probability. When the given allele increases its frequency, suddenly some other allele becomes more or less *desired* and the positive feedback becomes effective. Ideally, the SG during the random drift selects a local optimum, and the VP quickly evolves toward such a target.

To understand how the positive feedback is triggered, we can consider an allele $a_{i_1 j_1}$ that produces a good fitness only when allele $a_{i_2 j_2}$ is present in locus L_{j_2}. If allele $a_{i_2 j_2}$ randomly increases its frequency, then individuals with allele $a_{i_1 j_1}$ will more likely win tournaments, and allele $a_{i_1 j_1}$ will increase his frequency too. Positive feedback would quickly drive the VP towards a local optimum that includes both alleles $a_{i_1 j_1}$ and $a_{i_2 j_2}$. The convergence speed can be tuned using parameters ε_i: a high value will make the VP moving quickly towards the first good solution it finds, while a very small value will make the VP float for a longer time before choosing which local optimum to select as a target.

Figure 4 shows the behavior of the fitness in a typical run of the SG, as a function of the number of evaluations (calls to the fitness function). In the first calls, many random individuals are being evaluated, and their fitness is quite low. At this point, some trend casually emerges, and the SG focuses its search towards improving the solution in a hill-climbing fashion. The SG is actually different from a pure hill-climbing approach, since it keeps evaluating different neighborhood of different solutions: this can be seen by the "thickness" of the curve. The last phase of the search starts where the local minimum has been approached, and small variations are attempted to improve it slightly. The curve thickness reduces, to indicate that the variance of alleles in the gene pool is decreasing. This situation is indicated [Dawk89] as an *evolutionary stable strategy*, where small variations over the current set of individuals worsen their fitness, so that "mutants" can not invade the population. It is important to notice that this three-phase behavior is completely a consequence of evolution, and the SG is not aware of this behavior.

The behavior of the SG resembles, in this respect, that of *simulated annealing*: at the beginning it explores points of the solutions space that can be very far from each other; during the evolution process in each locus L_i one marginal probability will tend to 1, while the others will decrease to 0. Thus the explored points of the solution space will become closer and closer. Ideally, when all vectors $\mathbf{P}_i$ contain exactly one $p_{ij} = 1$, selected individuals differ only for the mutation effects.

The SG also resembles a *hillclimber*. GAs have the powerful ability to preserve potentially useful genetic material included in individuals that are not the fittest. On the contrary, the SG tends to discard all alleles that do no lead to the local optimum selected as a target. As in biological evolution [Dawk96], the path followed by the SG is always sloping upward. In fact, we expect that the SG is not suited to solve problems specifically designed to be difficult for hillclimbers, such as Holland's Royal Road Function [Holl93] [Jone93].

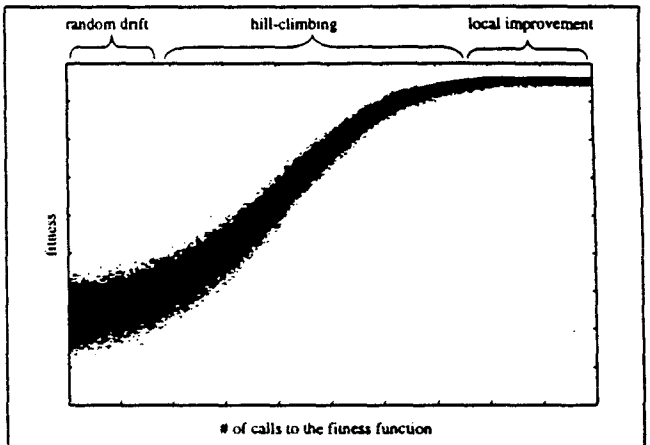

Figure 4: *Typical SG behavior*

For this reason, in order to guarantee sufficient exploration of the solution space, the SG should be iterated, in a multi-start fashion, to let it visit different local optima. At each iteration, a different random seed is used, the value of all ε_i are slightly reduced and all probabilities are set again to their initial values. Figure 5 reports a pseudo-code for the overall multi-start environment in which the SG is expected to be used. The algorithm is usually iterated until some stopping condition, related to the available computational resources (e.g., max CPU time or number of fitness function evaluations), is met.

[1] in the implementation, care needs to be taken to avoid any probability to become negative.

3.3. Algorithm Summary

Given the above definitions, the implementation of a SG for a given optimization problem is composed of the following steps:

- definition of the encoding of the solutions as genomes: this step is analogous to the one required for Genetic Algorithms. The main constraint, in this phase, is to be able to identify *a priori* the number of loci g and the possible n_i values for the alleles for each locus. Variable-length encoding schemes (where either g or the n_i's are not constant) have not been yet analyzed.
- definition of a fitness function, based on the evaluation of the goodness of a given individual. This function is exactly the same as in GA, where the evaluation takes place at the phenotypic level, after a decoding phase. Thanks to the tournament mechanism, no linearization is needed in SG.
- definition of the parameters driving the evolution: the positive feedback factors ε_i (experiments show that the optimal values do not depend on the number of alleles n_i in each locus, and are usually set to a value in the range 10^{-4}+10^{-2}); the mutation probability p_m (usually set to $1/g$ [Bäck93]); the steady-state threshold p_t (good values are typically over 0.95).
- implementation of the algorithm, where the probabilities **P** are evolved according to the proposed scheme. Memory allocation is related to this data structure, only, since no population needs to be stored. CPU time for the SG is mainly due to the procedure `select_individual`, which is often negligible with respect to the fitness computation.

```
genome multistart_SG()
{
  genome best, last ;
  best = SG() ;
  while(stopping_condition()==FALSE)
  {
    decrease_all(εi) ;
    last = SG() ;
    if( fitness(last)>fitness(best) )
      best = last ;
  }
  return best ;
}
```

Figure 5: *The SG in a multi-start environment*

4. Case Study: The 0/1 Multiple Knapsack Problem

To evaluate the effectiveness of the *Selfish Gene Algorithm* in solving combinatorial optimization problems, we applied the general framework to a case study, and compared the SG performance with other published results. We chose the 0/1 multiple knapsack problem.

4.1. Problem definition

In the *single-constraint 0/1 knapsack problem*, we are given a knapsack of capacity C and N objects. Each object has a weight w_i and a benefit (or profit) b_i (i=1...N). The optimization goal is to fill the knapsack with the set of objects that yield the maximum benefit. In other words, we aim at finding a vector $\mathbf{X} = (x_1, x_2, \ldots, x_N)$ where $x_i \in \{0, 1\}$, such that $\sum_{i=1\ldots N} w_i x_i < C$ and for which $\sum_{i=1\ldots N} b_i x_i$ is maximum. This is known to be a NP-complete problem and the partition problem can be polynomially transformed into it [GaJo79], and effective approximations have been developed for obtaining near-optimum solutions [MaTo90].

The *0/1 multiple knapsack problem* consists of M different knapsacks of capacities $C_1, \ldots, C_M$ and N objects, each of which has a benefit b_i. Unlike the single-constraint version, in which the weights of the objects are fixed, the i^{th} object weights w_{ij} when it is considered for inclusion in the j^{th} knapsack of capacity C_j. Once more, we are interested in finding a vector $\mathbf{X} = (x_1, x_2, \ldots, x_N)$ that guarantees that no knapsack is overfilled: $\forall j \in [1\ldots M] : \sum_{i=1\ldots N} w_{ij} x_i < C_j$, and that yields the maximum benefit $\sum_{i=1\ldots N} b_i x_i$. This problem is also known as the *zero-one integer programming problem* [GaJo79], and as the *0-1 linear programming problem* [MaTo90].

The problem can be thought as a resource allocation problem, where we have M resources (the knapsacks) and N objects. Each resource has its own budget (capacity), and w_{ij} represents the consumption of resource j by object i. Each object consumes a bit of all reources and we are interested in maximizing the profit, while working with a certain budget.

4.2. Prototypical implementation

A prototypical implementation of the SG has been developed, and amounts to about 500 ANSI C code lines. The prototype is able to evolve a generic VP with variable length list of alleles for each locus, and to implement the multi-start environment described in Section 3.2. In this prototypical implementation, the ε_i do not vary during the SG, but only across multi-start iterations.

This generic prototype has been used to solve the 0/1 multiple knapsack problem. The following sub-section details the definition of the SG, according to the general scheme of Section 3.3.

4.2.1. SG for the 0/1 Multiple Knapsack Problem

The adopted encoding is the most "natural one," where the genome is directly derived from the structure of the solutions $\mathbf{X} = (x_1, x_2, \ldots, x_N)$:

- *definition of the encoding*: the genome is a list of N loci (g=N), where each locus represents an object; two alleles are possible for each locus, stating whether the object is included in all the M knapsacks. Therefore we have n_i=2 for all loci L_i, with a_{i1}="present" and a_{i2}="absent".
- *definition of the fitness function*: the fitness func-

tion used in the SG is the same presented in [KBHe94]. Infeasible strings are allowed in the VP and a penalty term [RPLH89] is added to their fitness: the farther away from feasibility the string is, the higher the penalty term is. The fitness we adopt is:

$$f(X) = \sum_{i=1...N} b_i x_i - \max\{o_i\}\, s \qquad (2)$$

where s is the number of objects that overfill at least one knapsack.

- *definition of the values of parameters*: according to the general guidelines, we chose ε_i=0.01 for all i, and reduced it by a factor of 30% at each multi-start iteration. The probabilities are p_m=1/g=1/N and p_t=0.99.

problem instance				SG		
name	*N*	*M*	*maximum*	*average*	*best*	*CPU [s]*
FLEI	10	20	2,139	2,123.60	2,139	62.93
HP1	4	28	3,418	3,404.00	3,404	116.90
HP2	4	35	3,186	3,183.40	3,186	78.47
PB1	4	27	3,090	3,076.00	3,076	112.71
PB2	4	34	3,186	3,186.00	3,186	42.60
PB4	2	29	95,168	91,600.70	91,935	120.18
PB5	10	20	2,139	2,125.90	2,139	61.74
PB6	30	40	776	776.00	776	14.02
PB7	30	37	1,035	1,034.50	1,035	46.03
PET2	10	10	87,061	87,061.00	87,061	0.39
PET3	10	15	4,015	4,015.00	4,015	0.75
PET4	10	20	6,120	6,120.00	6,120	11.62
PET5	10	28	12,400	12,400.00	12,400	5.84
PET6	5	39	10,618	10,607.50	10,618	125.18
PET7	5	50	16,537	16,519.90	16,537	205.00
SENTO1	30	60	7,772	7,769.40	7,772	131.21
SENTO2	30	60	8,722	8,712.90	8,722	225.18
WEING1	2	28	141,278	141,278.00	141,278	12.67
WEING2	2	28	130,883	130,883.00	130,883	8.62
WEING3	2	28	95,677	95,542.50	95,677	94.25
WEING4	2	28	119,337	119,337.00	119,337	4.45
WEING5	2	28	98,796	98,719.50	98,796	86.92
WEING6	2	28	130,623	130,545.00	130,623	48.96
WEING7	2	105	1,095,445	1,095,316.20	1,095,445	411.55
WEING8	2	105	624,319	620,843.20	623,952	440.28
WEISH01	5	30	4,554	4,554.00	4,554	1.88
WEISH02	5	30	4,536	4,536.00	4,536	10.48
WEISH03	5	30	4,115	4,115.00	4,115	15.03
WEISH04	5	30	4,561	4,561.00	4,561	1.18
WEISH05	5	30	4,514	4,514.00	4,514	2.32
WEISH06	5	40	5,557	5,555.50	5,557	73.56
WEISH07	5	40	5,567	5,567.00	5,567	17.91
WEISH08	5	40	5,605	5,603.80	5,605	73.02
WEISH09	5	40	5,246	5,246.00	5,246	6.08
WEISH10	5	50	6,339	6,339.00	6,339	18.43
WEISH11	5	50	5,643	5,643.00	5,643	23.35
WEISH12	5	50	6,339	6,339.00	6,339	13.74
WEISH13	5	50	6,159	6,159.00	6,159	23.45
WEISH14	5	60	6,954	6,954.00	6,954	25.67
WEISH15	5	60	7,486	7,486.00	7,486	35.67
WEISH16	5	60	7,289	7,288.70	7,289	129.91
WEISH17	5	60	8,633	8,633.00	8,633	45.27
WEISH18	5	70	9,580	9,579.30	9,580	126.02
WEISH19	5	70	7,698	7,698.00	7,698	17.91
WEISH20	5	70	9,450	9,450.00	9,450	53.05
WEISH21	5	70	9,074	9,074.00	9,074	15.09
WEISH22	5	80	8,947	8,922.30	8,947	318.67
WEISH23	5	80	8,344	8,326.70	8,344	253.74
WEISH24	5	80	10,220	10,217.50	10,220	236.87
WEISH25	5	80	9,939	9,935.80	9,939	173.77
WEISH26	5	90	9,584	9,583.40	9,584	166.13
WEISH27	5	90	9,819	9,819.00	9,819	44.95
WEISH28	5	90	9,492	9,485.10	9,492	217.68
WEISH29	5	90	9,410	9,403.60	9,410	197.46
WEISH30	5	90	11,191	11,189.00	11,191	287.72

Table 1: *Experimental results*

4.3. Experimental evaluation

We performed two different evaluations of the SG algorithm: first, we ran our algorithm on a standard set of benchmarks; then, we compared the SG results with those of a GA solving the same problem.

4.3.1. Benchmarks results

The SG was evaluated on 55 test problems that are taken from the literature [Beas90]; problems range from 10 to 105 objects and from 2 to 30 knapsacks. Results are summarized in Table 1, whose left hand side (*problem instance*) reports the *name* of the problem, its size in terms of N and M, and the known *maximum* benefit that can be obtained.

For each test problem, a total of 10 runs of the multi-start procedure were executed. In each run, the stopping condition in `multistart_SG` was set to evaluate up to 200,000 genomes (i.e., individuals), unless the known optimum is reached before. All experiments were performed on a Sun SPARC Station 5/110 with 64 Mbytes of memory. The right hand side of Table 1 (*SG*) reports the *average* benefit and the *best* benefit value obtained by the SG over all 10 runs, and the average *CPU* time required for each run. In many cases the SG was able to reach the optimum solution in all 10 runs, and the *average* value is exactly equals to the *best* one.

4.3.2. Comparison with a GA

We compared the SG algorithm with a GA [KBHe94] able to solve the 0/1 multiple knapsack problem without any problem-specific knowledge. Test conditions reported in [KBHe94] are: 100 runs for each problem, a limit of 100,000 fitness evaluations for the first 7 problems and 200,000 evaluations for the last 2 problems. For the sake of comparison, we adopted exactly the same test conditions.

problem instance		[KBHe94]		GA	
name	*maximum*	*average*	*#max*	*average*	*#max*
knap15	4,015	4,012.70	83	4,015.00	100
knap20	6,120	6,102.30	33	6,119.90	99
knap28	12,400	12,374.70	33	12,400.00	100
knap39	10,618	10,536.90	4	10,594.43	22
knap50	16,537	16,378.00	1	16,493.59	11
sento1-60	7,772	7,626.00	5	7,763.79	54
sento2-60	8,722	8,685.00	2	8,709.72	25
weing7-105	1,095,445	1,093,897.00	0	1,095,310.42	8
weing8-105	624,319	613,383.00	6	620,228.97	0

Table 2: *Comparison with [KBHe94]*

Table 2 summarizes the results of the experiments. The first two columns (*problem instance*) report the *name* of the problem, taken from [KBHe94], and the *maximum* obtainable benefit. The following groups of columns report the results archived by [KBHe94] and by the SG, respectively. We show the *average* profit obtained over all 100 runs and, in the column *#max*, the number of times the best solution is reached.

Figure 6 plots the average profit as a percentage of the maximum profit.

Results show that the SG always produces a better result than [KBHe94]. Moreover, in [KBHe94] for solving the last problem the authors needed to heuristically bias the random initial population, while on the contrary for the SG no modification of any kind was required.

5. Conclusions

This paper proposes a novel optimization approach, the *Selfish Gene Algorithm*, inspired by the biological Selfish Gene Theory. The algorithm evolves a Virtual Population, in which alleles compete for appearance in their respective locus in the genotype. Competition is carried on at the phenotypic level by evaluating the fitness of one-shot individuals.

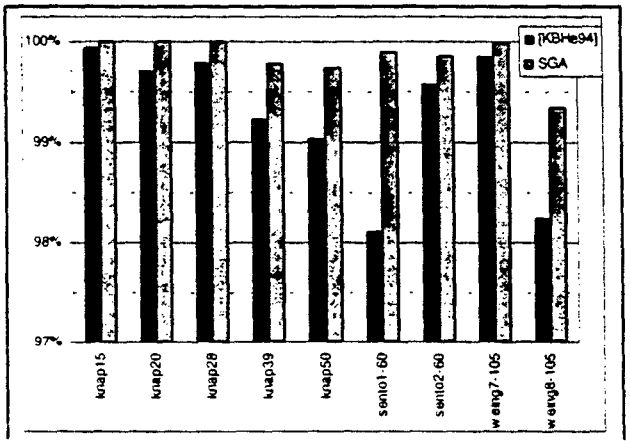

Figure 6: *Graphical summary of comparison with [KBHe94]*

The proposed approach is quite general, and relies mainly on the definition of a suitable encoding for genomes, and on a fitness function. If we compare the *Selfish Gene Algorithm* with Genetic Algorithms, we can claim that SG are easier to implement, have a smaller number of parameters to tune, do not rely on crossover operators, and are able to find more quickly optimal solutions. On the other hand, they tend to explore a smaller region of the search space, so they need to be inserted in some multi-start-like framework.

We claim that the *Selfish Gene Algorithm* is an interesting approach to be investigated as an optimization algorithm, as it compares favorably with other optimization approaches. More work is needed, and is currently being carried on by the authors, to clearly identify the subset of problems to which it is best applicable, and to evaluate the sensitivity to the various parameters.

6. References

[Bäck93] T. Bäck, *Optimal Mutation Rates in Genetic Search*, Proc. 5th International Conference on Genetic Algorithms and their Applications, 1993

[Beas90] J. E. Beasley, *OR-Library: Distributing Test Problems by Electronic Mail*, Journal of Operational Research Society, 41(11):1069-1072, 1990. All the files in OR-Library are available via anonymous ftp from `mscmga.ms.ic.ac.uk`

[Dawk82] R. Dawkins, "The Extended Phenotype", Oxford: W. H. Freeman, 1982

[Dawk89] R. Dawkins, "The Selfish Gene – new edition", Oxford University Press, 1989

[Dawk96] R. Dawkins, "Climbing Mount Improbable", W.W.Norton, New York and Viking Penguin, London, 1996

[GaJo79] M. R. Garey, D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", W. H. Freeman and Company, San Francisco, 1979

[Holl93] J. H. Holland, *Royal Road functions*, Internet Genetic Algorithm Digest, vol. 7, issue 22, August, 1993

[Jone93] T. Jones, *A Description of Holland's Royal Road Function*, Proc. 5th International Conference on Genetic Algorithm, 1993

[KBHe94] S. Khuri, T. Bäck, J. Heitkötter, *The Zero/One Multiple Knapsack Problem and Genetic Algorithms*, Proc. ACM Symposium on Applied Computation (SAC'94), 1994

[MaTo90] S. Martello, P. Toth, "Knapsack Problems: Algorithms and Computer Implementations", John Wiley & Sons, England 1990

[RPLH89] J. T. Richardson, M. R. Palmer, G. Liepins, M. Hilliard, *Some Guidelines for Genetic Algorithms with Penalty Function*, Proc. 3rd International Conference on Genetic Algorithm, 1989