On the Complexity of Hierarchical Problem Solving

Edwin D. de Jong
Institute of Information and Computing Sciences
Utrecht University
PO Box 80.089
3508 TB Utrecht, The Netherlands
dejong@cs.uu.nl

Richard A. Watson
School of Electronics and Computer Science Faculty of Engineering, Science and Mathematics
University of Southampton
SO17 1BJ Southampton
raw@ecs.soton.ac.uk

Dirk Thierens
Institute of Information and Computing Sciences
Utrecht University
PO Box 80.089
3508 TB Utrecht, The Netherlands
dirk@cs.uu.nl

ABSTRACT

Competent Genetic Algorithms can efficiently address problems in which the linkage between variables is limited to a small order k. Problems with higher order dependencies can only be addressed efficiently if further problem properties exist that can be exploited. An important class of problems for which this occurs is that of hierarchical problems. Hierarchical problems can contain dependencies between all variables ($k = n$) while being solvable in polynomial time. An open question so far is what precise properties a hierarchical problem must possess in order to be solvable efficiently. We study this question by investigating several features of hierarchical problems and determining their effect on computational complexity, both analytically and empirically. The analyses are based on the Hierarchical Genetic Algorithm (HGA), which is developed as part of this work. The HGA is tested on ranges of hierarchical problems, produced by a generator for hierarchical problems.

Categories and Subject Descriptors

F.2 [**Analysis of algorithms and problem complexity**]: General

General Terms

Algorithms, Theory, Experimentation, Performance

Keywords

Modularity, hierarchy, scalability, representation development

1. INTRODUCTION

A primary challenge in evolutionary computation is to increase the size of problems that can be addressed reliably. The feasible sizes of a problem strongly depend on the problem properties that can be exploited. In the absence of any such properties, there is no basis for reducing or biasing the search space, so that large problems cannot be addressed due to the exponential size of the search space.

A useful problem property that can render large problems feasible when identified is that of *dependencies*. Two variables in a problem are interdependent if the fitness contribution or optimal setting for one variable depends on the setting of the other variable. If all variables are independent, they can be optimized one by one, and the problem can be solved in linear time.

A somewhat more realistic assumption is that the *order* of the dependencies is limited to some small number k, e.g. $k = 2$ or $k = 3$, where the order is the largest number of variables that are interdependent. Such order-k separable problems can be solved reliably by Competent Genetic Algorithms (Goldberg, 2002); examples include the fast messy GA (Goldberg, Deb, Kargupta, & Harik, 1993), the gene expression messy GA (Kargupta, 1996), the linkage learning GA (Harik, 1997), the extended compact GA (Harik, 1999), the Bayesian Optimization Algorithm (BOA) (Pelikan, Goldberg, & Cantu-Paz, 1999), LFDA (Mühlenbein & Mahnig, 1999), and EBNA (Etxeberria & Larrañaga, 1999).

The assumption that all interactions in a large problem will be limited to a small number k is still a strong one. An important question therefore is which problems can be solved when many variables may be interdependent. We focus on a particular form of problem structure called *hierarchy*, which permits addressing certain large problems with high order dependencies, up to the case where all variables are interdependent ($k = n$).

Examples of hierarchical problems include HIFF (Watson, Hornby, & Pollack, 1998) and Hierarchical Trap Functions (Pelikan & Goldberg, 2001a). Methods that can solve certain hierarchical problems include SEAM (Watson & Pollack, 2003), H-BOA (Pelikan & Goldberg, 2001a, 2003), Compact Genetic Codes (Toussaint, 2005), and the *Hierarchical Genetic Algorithm* (HGA) (De Jong, Thierens, & Watson, 2004). The HGA is a new hierarchical method which has so far only been described theoretically (De Jong et al., 2004), but will be investigated in experiments here.

The feasibility of addressing certain hierarchical problems has been demonstrated. The question that will be studied here is *which* hierarchical problems can be addressed efficiently.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GECCO'05, June 25–29, 2005, Washington, DC, USA.

Copyright 2005 ACM 1-59593-010-8/05/0006 ...$5.00.

A definition of the class of hierarchical problems is provided in (De Jong et al., 2004). This definition employs several parameters. By varying these parameters, different hierarchical problems are specified. These parameters therefore provide a way to study the space of hierarchical problems. To this end, we have developed a generator for hierarchical problems, which is described in detail elsewhere (De Jong, Watson, & Thierens, 2005).

Our goal in this paper is twofold. First, we aim to study the effect of the parameters that characterize hierarchical problems on the computational requirements for these problems. Second, we aim to investigate the operation of the HGA. To address these goals, we apply the HGA to several ranges of hierarchical problems, and study the effect of the parameters on computational expense, measured both in evaluations and in runtime. The problems are obtained by means of a generator for hierarchical problems. The Hierarchical Problem Generator is available from the homepage of the first author.

The structure of this paper is as follows. The introduction is followed by a description of the class of hierarchical problems (Section 2) and of the generator for hierarchical problems that will be employed (Section 3). Next, Section 4 describes the implementation of the HGA that will be employed in detail. Section 6 reports the experiments, and investigates the effect of the different problem parameters. Finally, a discussion is provided (Section 7) and conclusions are drawn (Section 8).

2. THE CLASS OF HIERARCHICAL PROBLEMS

A formal delineation of the class of hierarchical problems was first given in (De Jong et al., 2004).[1] Below, an informal description of the problem class is given.

Watson (2002) defines *decomposability* as the property that the number of optimal settings for a module is lower than its total number of settings, and provides an extensive discussion of this and related ideas. Based on this concept, the notions of modularity and hierarchy can be defined.

A *module* is a subset of the variables in a problem for which it holds that only *part* of the variable settings are near-optimal for some context setting, or *ϵ-context-optimal*. Here a *context setting* is a setting for the remaining variables, i.e. the variables not contained in the module. A variable setting is *near-optimal* given a context setting if the fitness it results in is at most a small value ϵ less than the highest fitness that can be obtained given the context setting. Since only part of the possible settings for a module are ϵ-context-optimal, the remaining settings can be safely excluded from the search. Thus, the detection of a module reduces the computational expense required to address the problem.

Hierarchy applies the modularity principle recursively and thereby permits reductions at increasingly large scales. This makes it possible to address certain problems with high order linkage in a scalable manner.

More precisely, hierarchy is defined as follows. Let the *primitive modules* in a problem consist of the problem variables. A set of existing modules can be combined into a *composite module* if and only if the number of ϵ-context-optimal settings for the combination is lower than the product of the numbers of context-optimal settings for the components, and no subset of the module combination establishes such a reduction.

Given these notions, a *modular problem* can be defined as a problem that contains at least one composite module. Hierarchy refers to the presence of modules at multiple levels above the base layer of primitive modules. Thus, a *hierarchical problem* can be defined as a problem in which there is at least one composite module that contains another composite module.

A problem is called *fully hierarchical* if there exists a single module that covers all indices and whose settings thus specify complete individuals; such a module will be called a *maximal module*. To indicate the complexity of a fully hierarchical problem, it is useful to determine the maximum number k of component modules that must be considered for combination into a larger module. A problem will be called *order-k fully-hierarchical* if, starting from the initial level of primitive modules, recursively combining up to k modules into larger modules will lead to identification of the maximal module.

3. A GENERATOR FOR HIERARCHICAL PROBLEMS

Based on the definition of the class of hierarchical problems, we have developed a generator for hierarchical problems which will now be discussed; for a detailed account the reader is referred to (De Jong et al., 2005).

A hierarchical problem is characterized by the following factors:

- n: the number of variables
- k: the maximum number of components of a module
- m: the maximum number of context-optimal settings for a module
- s: the arity of the alphabet (e.g. s=2 for binary problems).

The Hierarchical Problem Generator (HPG) accepts values for the above parameters and randomly generates a hierarchical problem with these properties. The operation of the HPG is as follows. First, a module set is initialized to contain all primitive modules. Next, the following cycle is repeated until the module set contains a single module:

Combinations of k modules are selected randomly without replacement from the module set, until the set is exhausted. Each combination will form a new module, and the resulting modules constitute the new module set.

For each new module, m context-optimal settings are chosen randomly. Each context-optimal setting is a combination of context-optimal settings for the components of the module.

All context-optimal settings of a module at level r are used at least once in a context-optimal setting at level $r + 1$. Employing this principle recursively ensures that the chosen context-optimal settings of a module at the lowest level will occur at higher levels, up to the highest level of the maximal module, and will therefore occur in one of the global optima. This guarantees that the chosen settings will indeed be context-optimal as intended.

[1]While generators of hierarchical problems have been described in the literature, these are not restricted to generating hierarchical problems, and thus do not serve to delineate the problem class.

find-cosettings(m, nrsettings)

1. **for** $i := 1 : |S| \quad \wedge \quad (|m.\text{cos}| \; < \text{nrsettings}))$
2. $\quad f_{max}$:=find-best-fitness(m,samples[i]);
3. $\quad m$.cos:=m.cos $\cup$ find-settings(m, sample[i], f_{max});
4. **end**

Figure 1:

The above description implies that the HPG generates order-k fully-hierarchical problems; we believe these to be most relevant in studying the computational properties of hierarchical problems. The above choices can be varied however to generate a larger set of hierarchical problems. For example, the following variations may be considered:

- Unbalanced trees. As given, the HPG produces fully hierarchical problems. If module formation is interrupted below the top level of the tree, some degree of the hierarchical structure can still be exploited, while identifying the global optima may no longer be computationally feasible.
- Variable k. In the above implementation of the HPG, all modules contain the same number of components k. This number could be varied, for example by selecting a random number $k' \leq k$ of components for each module.
- Variable m. The HPG assigns the same number m of context-optimal settings to each module. A variation would be to assign a variable number $m' \leq m$ of context-optimal settings to each module.
- Variable s. All variables are assumed to come from the same alphabet. A possible variation is to permit different alphabets for different variables.

The above variations, and all combinations thereof, provide a means to extend the class of hierarchical problems generated by the HPG. In the following section, we describe the HGA, which will be used to address hierarchical problems.

4. THE HIERARCHICAL GENETIC ALGORITHM

The Hierarchical Genetic Algorithm framework was described theoretically in (De Jong et al., 2004). The idea of the algorithm is as follows. The methods starts from a set of *primitive modules* which equals the set of variables in the problem. Next, the following cycle is repeated. For any combination of up to k existing modules, test whether the combination is a module itself. If so, the module is formed.

Module formation works as follows. Given a candidate module, it must be tested whether the number of context-optimal settings is smaller than the number of possible settings. If this is the case, the module is formed and replaces its component modules, and the context-optimal settings for the new module are stored.

The contexts are generated by sampling; the sampling procedure takes the current set of modules and for each

H-GA()

1. $t := 0$;
2. $\mathcal{M}_t$:=find-primitive-modules();
3. **for** $i := 1 : |S|$
4. $\quad$ samples[i]:=find-sample($\mathcal{M}$);
5. done:=false;
6. **while** ($\neg$ done)
7. $\quad \mathcal{M}_{t+1}$:=module-formation($\mathcal{M}_t$);
8. $\quad$ **if** $\mathcal{M}_{t+1} = \mathcal{M}_t$
9. $\quad\quad$ done:=true;
10. $\quad t++$;
11. **end**
12. Construct-Solution($\mathcal{M}$);

Figure 2: Main loop of the HGA; see text.

module-formation($\mathcal{M}$)

1. $k' := 2$;
2. formed:=false;
3. **while** $k' \leq k \; \wedge \; \neg$formed{
4. $\quad$ **if** $k' > |\mathcal{M}|$
5. $\quad\quad$ **return** true;
6. $\quad$ **forall** $mc \in \mathcal{M}^{k'} \; \wedge \; \neg$formed
7. $\quad\quad$ nrsettings := $\prod_{m \in mc} |m.\text{cos}|$
8. $\quad\quad$ find-cosettings(mc, nrsettings);
9. $\quad\quad$ **if** $|mc.\text{cos}| <$ nrsettings
10. $\quad\quad\quad \mathcal{M}' := mc \, \cup \, \mathcal{M} \, \setminus \, \{m | m \in mc\}$
11. $\quad\quad\quad$ update-samples(mc);
12. $\quad\quad\quad$ formed:=true;
13. $\quad\quad$ **end**
14. $\quad$ **end**
15. **end**
16. $k'++$;
17. **return** $\mathcal{M}'$;

Figure 3:

module randomly selects one of the context-optimal settings. Since sampling only uses the context-optimal settings of modules, the formation of a module has the effect that all non-context-optimal settings for the module will be excluded for the remainder of the search. The number of excluded settings grows quickly over time, and this is how the method operates; if the method makes correct decisions, which can be guaranteed to any desired degree by choosing a sufficiently large sample size, the module set will eventually contain only a single module, and any context-optimal settings for this maximal module are global optima.

We describe an implementation of the HGA framework described in (De Jong et al., 2004). The pseudocode for the algorithm is shown in Figures 2, 3, and 1. The algorithm closely follows the description in (De Jong et al., 2004), but two ideas have been added: first, the samples that are used to assess the validity of a module are not discarded, but maintained so that they can be used again in the future. All fitness values are cached, so that unnecessary double evaluations are avoided. The initial samples can however not be used indefinitely; if a new module is formed for which the excluded settings occur in some samples, then these samples must be discarded. Therefore, a check is performed every time a new module is formed. The procedure **update-samples** discards such invalidated samples, and fills the set of samples with new samples until it reaches its original size again.

A second new element of the algorithm is that in finding the context-optimal settings of a candidate module (See Figure 1), the algorithm does not necessarily consider all sample points; as soon as all possible settings for the module have been identified as being context-optimal, no further sample points need to be considered. The following functions are used:

find-best-fitness(m**, sample)** finds the maximum achievable fitness for a combination of modules given the context setting defined by `sample`.

find-settings(m**, sample,** f**)** finds all settings for m, using *sample* to define the remaining variables, that yield a fitness within ϵ of fitness value f.

4.1 Determining the Sample Size Analytically

One of the main choices that must be made when applying the HGA concerns the sample size. The sample size must be sufficient to detect all context-optimal settings of a candidate module; if some context-optimal settings of an incorrect module combination are not discovered, the combination will appear to be a module, and the missing settings will be excluded for the remainder of the search. Choosing the sample size higher than necessary does not affect the correctness of the algorithm's operation, but results in superfluous computational expense.

A question of theoretical interest is whether the sample size required to achieve correctness with sufficient probability can be calculated. In the following, we will derive an analytical expression that addresses this issue.

We consider the case where an incorrect module combination is tested for modularity. The goal is to choose the sample size such that with sufficient probability, all context-optimal settings, which are given by the product of the context-optimal settings of the components, will be found.

First, we consider the probability that given a single sample point, a particular context-optimal setting will be found. Since a module combination consists of at most k components, the components of an incorrect module combination mc can form part of at most k modules overlapping with mc. Each of these overlapping modules has at most m context-optimal settings. Therefore the probability that a particular setting for an element of mc will be context-optimal for a given context is $\frac{1}{m}$. The probability of finding a specific context-optimal setting for mc therefore equals $p_{\text{find}} = \frac{1}{m}^k$.

The probability of not finding a particular context-optimal setting given one sample point is $p_{\text{miss}} = 1 - p_{\text{find}} = 1 - \frac{1}{m}^k$. The probability that a particular context-optimal setting is not found for any sample point is $p_{\text{miss}}^{|S|} = (1 - \frac{1}{m}^k)^{|S|}$. Thus, the probability of not missing the context-optimal setting is $P_{\text{detect}} = 1 - (1 - \frac{1}{m}^k)^{|S|}$.

The probability of finding all m^k context-optimal settings at least once given $|S|$ sample points is ${P_{\text{detect}}}^{m^k}$, and this gives the probability that an incorrect candidate module is revealed to be incorrect. The probability that every incorrect candidate module considered before encountering a correct module is revealed to be incorrect is ${P_{\text{detect}}}^{m^k n^k}$. Thus, the probability that all modules in the problem are found without constructing incorrect modules is ${P_{\text{detect}}}^{m^k n^k n_m}$, where n_m is the number of modules in the problem. Finally, to achieve correct results for every run, the probability is ${P_{\text{detect}}}^{m^k n^k n_m n_r}$, where n_r is the number of runs.

In order to use the above expression, the number of modules in the problem must be determined.

At the lowest level, the n variables of the problem can be combined into $\frac{n}{k}$ modules. At the next level, the resulting modules can be combined into $\frac{n}{k^2}$ modules. Continuing like this, the number of modules in the problem equals

$$\begin{aligned} &\frac{n}{k} + \frac{n}{k^2} \ldots + \frac{n}{k^{log_k n}} \\ &= n(\frac{1}{k} + \frac{1}{k^2} \ldots + \frac{1}{k^{log_k n}}) \end{aligned}$$

This expression can be simplified using the following geometric series: $\frac{1-\frac{1}{k}^{n+1}}{1-\frac{1}{k}} = 1 + \frac{1}{k} + \frac{1}{k^2} .. + \frac{1}{k^n}$ Using this equality, we obtain the following:

$$\begin{aligned} n(\frac{1-\frac{1}{k}^{log_k(n)+1}}{1-\frac{1}{k}} - 1) &= \\ n(\frac{1-\frac{1}{kn}}{1-\frac{1}{k}} - 1) &= \\ n(\frac{kn-1}{kn-n} - 1) &= \\ n(\frac{n-1}{kn-n}) &= \\ \frac{n-1}{k-1} & \end{aligned}$$

Given this expression for the number of modules n_m, we can determine the required sample size as follows. Let

$$\begin{aligned} a &= 1 - m^{-k} \\ b &= m^k * n^k * n_m * nr \\ n_m &= \frac{n-1}{k-1} \end{aligned}$$

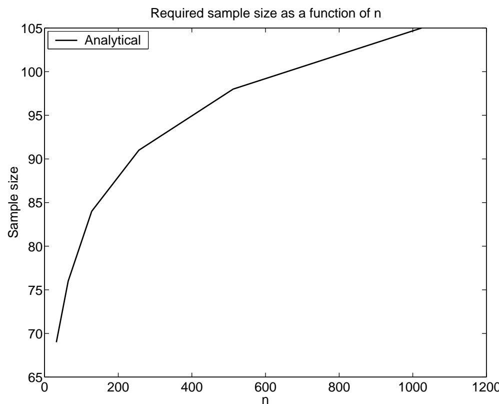

Figure 4: Analytical sample size for different problem sizes.

Then the probability of achieving correct results for each module in every run equals $p_{\text{correct}} = (1 - a^{|S|})^b$. This expression can be used to determine the required sample size as follows:

$$\begin{aligned} \ln(p_{\text{correct}}) &= b\ln(1 - a^{|S|}) \\ \ln(1 - a^{|S|}) &= \frac{\ln(p_{\text{correct}})}{b} \\ 1 - a^{|S|} &= e^{\frac{\ln(p_{\text{correct}})}{b}} \\ a^{|S|} &= 1 - e^{\frac{\ln(p_{\text{correct}})}{b}} \\ |S| &= \frac{\ln(1 - e^{\frac{\ln(p_{\text{correct}})}{b}})}{\ln(a)} \\ &= \frac{\ln(1 - e^{\frac{\ln(p_{\text{correct}})}{m^k * n^k * n_m * nr}})}{\ln(1 - m^{-k})} \end{aligned}$$

As an example, let us consider a 64-bit problem with $k = m = 2$. Then the sample size required to obtain correct results for each of 30 runs with probability $p_{\text{correct}} = .99$ equals 76.

The required sample size depends linearly on the number of level in the problem, or logarithmically on the problem size, as seen in Figure 4. We will now discuss the assumptions made by the above derivation. The expression is optimistic in that it assumes that the true modules with which an incorrect module overlaps have been formed up to the same level as the candidate module's components. On the other hand, it is pessimistic in that it assumes that all combinations of k out of n elements must be considered; this is only true in the worst case at the lowest level. A further pessimistic assumption is that all parts of an incorrect module are part of different module.

5. COMPUTATIONAL COMPLEXITY

We now turn to the computational complexity of the Hierarchical Genetic Algorithm that has been described, measured in terms of the number of fitness evaluations. De Jong et al. (2004) establish an upper bound for the HGA of $n_m k n_m^k m^k |S|$. In the following, we will derive a more precise expression for the expected number of evaluations. To this end, we first consider the expected amount of evaluations required to form the modules of the lowest level.

In a fully hierarchical problem where all modules have the same number of components k, the algorithm will first consider smaller combinations from $k' = 2$ up to $k' = k - 1$ modules. Since these do not lead to the construction of modules when correct decisions are made, we first consider the final iteration of $k' = k$.

Since all variables are part of some module in a fully hierarchical problem, the first component of the first module to be formed may be chosen arbitrarily; the algorithms visits the modules it maintains in order, and will thus select the first variable for this purpose. Let us say this variable is part of a module M. Then the number of modules that must be tested before the remaining component of M are found is given by the number of ways in which $k-1$ of the remaining $n-1$ variables can be chosen, i.e. $\binom{n-1}{k-1}$. For the remaining modules at the first level, the number of candidate modules is lower than this. Since the number of modules at the first level equals $\frac{n}{k}$, the number of candidate modules that must be considered to form the modules of the first level is at most $\frac{n}{k}\binom{n-1}{k-1}$.

However, before the case of $k' = k$, the algorithm must consider lower values of k' since the true value of k is assumed unknown. For $k' = k - 1$, all combinations of $k - 1$ modules must be considered; this number is slightly higher than for $k' = k$, namely $\binom{n}{k-1}$. For lower values of k, the number is lower. Therefore, the total number of module combinations that must be considered to form modules in the first layer is less than $(k-1)\binom{n}{k-1}$. For the remaining layers, the number of modules that may be combined is smaller than for the first layer. Thus, forming all modules involves considering at most $n_m(k-1)\binom{n}{k-1}$ modules combinations.

For each candidate module, the number of complete individuals that must be evaluated is at most $m^k|S|$. Thus, an upper bound for the time complexity of the algorithm is

$$\begin{aligned} & n_m(k-1)\binom{n}{k-1}m^k|S| \\ = & \frac{n-1}{k-1}(k-1)\binom{n}{k-1}m^k|S| \\ < & (n-1)\frac{n^{k-1}}{(k-1)!}m^k|S| \\ = & \frac{n^k - n^{k-1}}{(k-1)!}m^k|S| \end{aligned}$$

For the case of $k = 2$, this amounts to $(n^2 - n)m^k|S|$.

6. RESULTS

In the following, we describe experimental results with the HGA. The main question of interest is its computational complexity. Therefore, we will investigate how the amount of computational expense varies with properties of the problems considered. The problems that will be used are produced using the HPG.

When considering the computational complexity of an algorithm, two measures are of interest: the number of evaluations spent and the overall number of operations. The number of evaluations facilitates easy comparison with methods from the literature, and is of interest as the fitness calculation is often thought to be an expensive step in the algo-

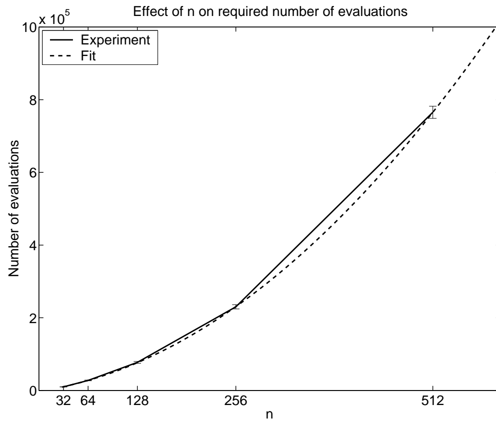

Figure 5: Computation time in number of evaluations as a function of n.

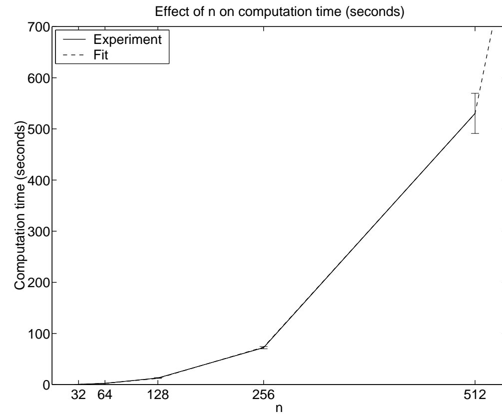

Figure 6: Computation time in CPU seconds as a function of n.

rithm. However, when applying a method, the actual number of operations is the more relevant factor, and furthermore the cost of certain algorithms is determined more by the operation of the algorithm itself than by the calculation of fitness values. We therefore consider both factors; which factor is more relevant for a given application will depend on the balance between the amount of time required by the fitness calculation and by the rest of the algorithm.

6.1 Scalability: effect of n

The main factor of interest when evaluating an algorithm is its scalability, i.e. the relation between computation time and the size of the problem, typically measured in terms of the number of variables n. To study this relation, random hierarchical problems were generated for n=32, 64, 128, 256, 512. The remaining parameters for the problem generator are : $k = 2$ and $m = 2$, where k is the number of components per module and m is the number of context-optimal settings per module. The HGA has parameters $k = 2$.[2]

The sample size $|S|$ was chosen empirically to be sufficient for making correct decisions only, and set to 100. Since the order in which candidate modules are considered is deterministic, when only correct decisions are made the algorithm is deterministic. Therefore, we average the performance of the method over 30 instances of the hierarchical problem for each combination of parameters.

6.1.1 *Number of evaluations*

Since the expression found for the complexity of the algorithm is quadratic in n, the number of evaluations found in the experiments was fit to the analytical expression $an^2 + bn + c$, yielding the parameters $a = 2.30, b = 321$, and $c = -2377$. As Figure 5 shows, this expression fits the empirical data very well.

6.1.2 *CPU time*

We have analyzed the total runtime of the HGA as a function of n. find-primitive-modules() costs $O(n|S|mftn(n))$, where $ftn(n)$ is the amount of time taken up by the fitness function for problem size n. The central loop of the algorithm is the module formation loop. The complexity for this loop is given by multiplying the expression for the number of fitness evaluations by the amount of time required to perform a fitness evaluation, which can be written as $ftn(n$. Thus, for $k = 2$, the total complexity of the algorithm amounts to $(n^2 - n)\ m^k |S| ftn(n)$.

For extra precision, the linear term will be given a parameter of its own, so that we need to fit the runtime data to $(an^2 + bn + c)ftn(n)$. To this end, we divide the measured runtimes by averaged values of $ftn(n)$ obtained by calculating the fitness of 1000 randomly generated individuals, and multiply by the same values after fitting. The results are shown in Figure 6. The fitted quadratic curve fits the data very well, so that the fitted curve is difficult to distinguish in the figure.

For all experiments except $n = 512$, a sample size of 50 is also sufficient to obtain correct results for all runs. Using this sample size, the 256-bit version of the problem is solved in approximately half a minute on a regular PC.

6.2 Effect of m

We measured the computational complexity as a function of m, the number of context-optimal settings per module. The dependency of the complexity of the algorithm on m is given by $a * m^k + b$; thus, the HGA is polynomial in m. While the number of components per module k used by the generator was 3, all instances of these problems were solved by the HGA using $k = 2$. Therefore, we used both $k = 2$ and $k = 3$ to compare the data with the above analytical expression. Further parameters are $n = 81$ and $|S| = 50$. In order to obtain a consistent value of m for all levels, the alphabet size s was set to equal m.

As Figure 7 shows, a closer fit is obtained using $k = 3$. A possible explanation for this is that while all modules could be found using only two components, the number of modules required to address the problem in this manner is larger than for a balanced binary decomposition; for example, when modules of size 2 and 1 are combined to form a size 3 module, the two module formation steps have only combined 3 variables, as compared to 4 in the binary case.

[2]We note that a larger k would have no effect, as the module formation step first considers combinations of $k' = 2$ modules and the algorithm terminates when the number of remaining modules is less than k'.

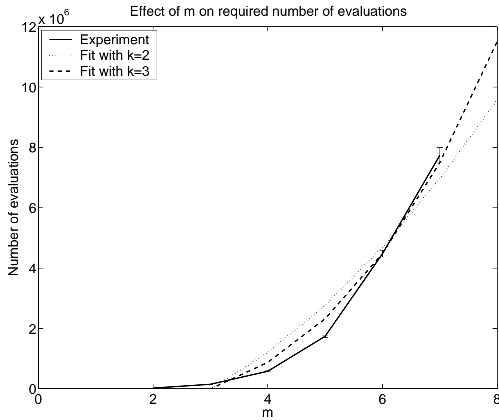

Figure 7: Dependency of the number of evaluations on m.

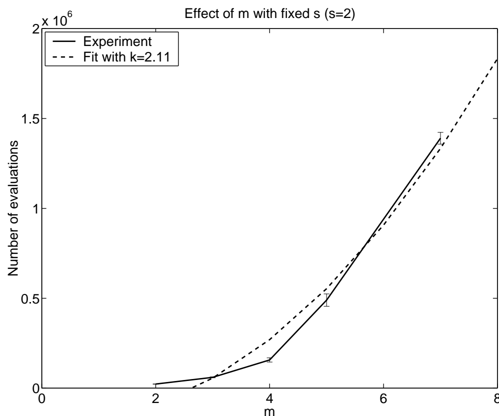

Figure 8: Dependency of the number of evaluations on m using a fixed alphabet size of $s = 2$.

In a further experiment, we considered the effect of using a binary alphabet ($s = 2$) for the same experiments, see Figure 8. As a result of this change, the modules in the problem can no longer all be found by considering combinations of 2 modules only; the HGA sometimes needs to combine $k = 3$ components into a module. The average number of components for modules formed by the methods was $k = 2, 2, 2.01, 2.03, 2.51$ for $m = 2, 3, 4, 5, 7$ respectively.

For $m = 4$, in two runs the HGA identified modules that cannot be combined into larger modules without considering values of k larger than three. Such dead-end modules can for example arise when two components of a module already establish a reduction, and thus represent a correct module, while the addition of the third variable does not establish a further reduction, and will thus not be performed. If this occurs for three size-3 modules that form part of a size-9 module, identifying the size-9 module would require considering combinations of 3 size-2 and 3 size-1 equals $k = 6$ modules.

While a correct composition of these problems exists for which $k = 3$, the problems are not order-k fully hierarchical. In order to limit the investigation to true order-k fully

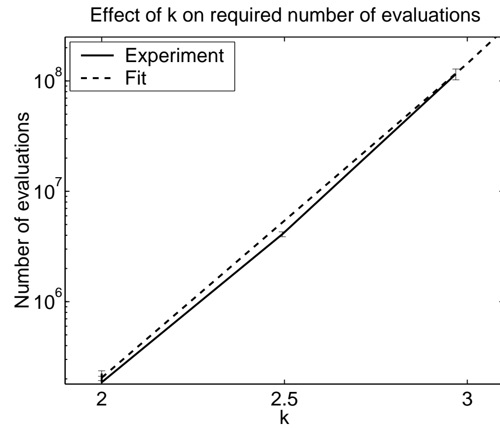

Figure 9: Dependency of the number of evaluations on k.

hierarchical problems therefore, two additional runs were performed to replace the runs concerned. For the case of $m = 6$, dead-end module arose in many of the runs; this datapoint is therefore omitted. For other values of m, including $m = 7$, the problem did not occur; even if a size-2 and a size-1 module cannot be joined together, they can sometimes be joined together with a size-3 or a size-6 module to a size-6 or size-9 module that does establish a further reduction of the number of context-optimal settings. The variations in the effective value of k may explain why the fit of the curve is not very close.

6.3 Effect of k

Finally, we consider the effect of k on the computational complexity. The expression derived for the complexity of the HGA shows that the dependency on k is exponential. We therefore fit the experimental data to the expression $ab^k + c$.

Experiments were run for $k = 2, 3, 4, 5$. The effective average values of k used by the HGA in these experiments were $k = 2, 2, 2.49, 2.97$ respectively. Figure 9 show the results; a reasonably close fit is obtained.

7. DISCUSSION AND RELATED WORK

Four factors have been investigated that determine the complexity of hierarchical problem solving: n, m, k, and $|S|$. It was seen that the time required to address fully hierarchical problems is polynomial in n and m, and exponential in k. For the required sample size, an analytical expression has been derived.

An upper bound of $\frac{n^k - n^{k-1}}{(k-1)!} m^k |S|$ has been established for the complexity of the HGA when measured in terms of the number of fitness evaluations.

Two existing algorithms that can reliably address certain hierarchical problems are hBOA and SEAM. For SEAM, an upper bound of $O(n^2 \ln n)$ has been established for the case of $m = k = 2$. For this case, the HGA has a complexity of $O(n^2 - n)$.[3] A main distinction between SEAM and the HGA is that whereas the HGA combines variables, SEAM

[3]As in the SEAM analysis, the sample size $|S|$ can be assumed constant here since the accuracy of sampling is independent of n.

combines variable settings. For problems where $m > s$, a module setting at one level may need to feature in more that one module at the next level. In order to address such problems, the initial module supply of SEAM can be increased, although this increases the computational requirements of the method. Conversely, considering the combination of variables settings rather than variables may permit addressing certain problems with overlapping modules.

For the hBOA algorithm, a bound of $O(n^{1.5}\log\ m$ has been proposed (Pelikan & Goldberg, 2001b). This bound is for problems of bounded difficulty; an interesting question is how the complexity of hBOA relates to our variable k.

hBOA employs a model building step that requires $O(n^3 + n^2 N)$ operations (Pelikan et al., 1999), where N is the number of instances and the maximal number of incoming edges k is assumed constant. Depending on the complexity of the fitness function, this procedure, which must be performed at every generation, may have more impact on the overall complexity than the bound for the number of evaluations.

The overall complexity of the HGA is reflected by the runtime of the algorithm in terms of CPU time, which was discussed above. Based on this, we expect that the HGA can be more efficient than existing methods for hierarchical problems where the fitness function itself is not overly expensive. A relevant direction for future research is to analyze the conditions that determine this by means of analytical and/or comparative experimental studies. The simplicity of the HGA is another property that may make it a useful candidate for addressing hierarchical problems.

8. CONCLUSIONS

We have investigated four central properties of hierarchical problems: the number of variables n, the number of components per module k, the number of context-optimal settings per module m, and the sample size $|S|$ used to address the problem. The influence of these factors on the computational complexity of hierarchical problem solving has been investigated both analytically and empirically based on the Hierarchical Genetic Algorithm (HGA). As part of this investigation, a generator of hierarchical problems has been developed and described.

Analytical expressions for the dependency of the complexity on the different factors have been derived. It was found that hierarchical problems can be solved in time polynomial in both n and m, but depend exponentially on k. The findings are confirmed by empirical data. The sample size required to achieve correct results with a fixed probability p depends logarithmically on n.

The Hierarchical Genetic Algorithm is a fast, simple hierarchical genetic algorithm; empirical measurements were consistent with an overall complexity of $O(n^2)$, and 256-bit problems were solved correctly in approximately half a minute on a regular PC. Interesting possibilities for future work include investigating the properties and feasibility of real-world hierarchical problems, the application of the HGA to such problems, and the further development of efficient hierarchical genetic algorithms.

Acknowledgments

The authors would like to thank Peter Bosman and Olwijn Leeuwenburgh for helpful suggestions.

References

De Jong, E. D., Thierens, D., & Watson, W. (2004). Hierarchical genetic algorithms. In Yao, X., Burke, E., Lozano, J. A., Smith, J., Merelo-Guervós, J. J., Bullinaria, J. A., Rowe, J., Tiňo, P., Kabán, A., & Schwefel, H.-P. (Eds.), *Parallel Problem Solving from Nature - PPSN VIII*, Vol. 3242 of *LNCS*, pp. 232–241, Birmingham, UK. Springer-Verlag.

De Jong, E. D., Watson, R. A., & Thierens, D. (2005). A generator for hierarchical problems. In Toussaint, M., Wright, A. H., & De Jong, E. D. (Eds.), *Proceedings of the GECCO 2005 Workshop on Theory of Representations*.

Etxeberria, R., & Larrañaga, P. (1999). Global optimization using bayesian networks. In Rodriguez, A. O., Ortiz, M. S., & Hermida, R. S. (Eds.), *Proceedings of the Second Symposium on Artificial Intelligence CIMAF*.

Goldberg, D. E. (2002). *The design of innovation. Lessons from and for competent genetic algorithms.* Kluwer Academic Publishers.

Goldberg, D. E., Deb, K., Kargupta, H., & Harik, G. (1993). Rapid, accurate optimization of difficult problems using fast messy genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pp. 56–64.

Harik, G. (1997). *Learning gene linkage to efficiently solve problems of bounded difficulty using genetic algorithms.* Ph.D. thesis, University of Michigan.

Harik, G. (1999). Linkage learning via probabilistic modeling in the ECGA. Tech. rep. Illigal report no. 99010, University of Illinois at Urbana-Champain, Urbana, IL.

Kargupta, H. (1996). SEARCH, evolution, and the gene expression messy genetic algorithm. Tech. rep. LA-UR 96-60, Los Alamos National Laboratory.

Mühlenbein, H., & Mahnig, T. (1999). FDA - A scalable evolutionary algorithm for the optimization of additively decomposed functions. *Evolutionary Computation*, *7*(4), 353–376.

Pelikan, M., & Goldberg, D. E. (2001a). Escaping hierarchical traps with competent genetic algorithms. In Spector et al., L. (Ed.), *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-01*, pp. 511–518. Morgan Kaufmann.

Pelikan, M., & Goldberg, D. E. (2001b). Hierarchical bayesian optimization algorithm = bayesian optimization algorithm + niching + local structures. In *Optimization by Building and Using Probabilistic Models (OBUPM) 2001*, pp. 217–221, San Francisco, California, USA.

Pelikan, M., & Goldberg, D. E. (2003). A hierarchy machine: learning to optimize from nature and humans. *Complexity*, *8*(5), 36–45.

Pelikan, M., Goldberg, D. E., & Cantu-Paz, E. (1999). BOA: The bayesian optimization algorithm. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., & Smith, R. E. (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, Vol. 1, pp. 525–532, San Francisco, CA. Morgan Kaufmann.

Toussaint, M. (2005). *Foundations of Genetic Algorithms 8 (FOGA 2005)*, chap. Compact genetic codes as a search strategy of evolutionary processes. Morgan Kauffman.

Watson, R. A. (2002). *Compositional Evolution: Interdisciplinary Investigations in Evolvability, Modularity, and Symbiosis.* Ph.D. thesis, Brandeis University.

Watson, R. A., Hornby, G. S., & Pollack, J. B. (1998). Modeling building-block interdependency. In Eiben, A., Bäck, T., Schoenauer, M., & Schwefel, H.-P. (Eds.), *Parallel Problem Solving from Nature, PPSN-V.*, Vol. 1498 of *LNCS*, pp. 97–106, Berlin. Springer.

Watson, R. A., & Pollack, J. B. (2003). A computational model of symbiotic composition in evolutionary transitions. *Biosystems*, *69*(2-3), 187–209. Special Issue on Evolvability, ed. Nehaniv.