Genetic State-Space Search for Constrained Optimization Problems

Jan Paredis
Research Institute for Knowledge Systems
PO Box 463, NL-6200 AL Maastricht, The Netherlands

Abstract

This paper introduces GSSS (Genetic State-Space Search). The integration of two general search paradigms — genetic search and state-space-search - provides a general framework which can be applied to a large variety of search problems. Here, we show how GSSS solves constrained optimization problems (COPs). Basically, it searches for "promising search states" from which good solutions can be easily found. Domain knowledge in the form of constraints is used to limit the space to be searched. Interestingly, our approach allows the handling of constraints within genetic search at a general domain independent level.

First, we introduce a genetic representation of search states. Next, we provide empirical results which compare the relative merit of the introduction of constraints during the generation of the initial population, during the fitness calculation, and during the application of genetic operators. Finally, we describe some extensions to our method which came about when applying it to factory floor scheduling problems.

1. Introduction

A large number of problems originating from different fields of study (e.g. economics, engineering) belong to the class of COPs. As a result, research into efficient solution methods for COPs has become a field of study in its own right.

Constrained optimization problems typically consist of a set of n variables xi (i < n) which have an associated domain Di of possible values. There is also a set C of constraints which describe relations between the values of the xi. (e.g. the value of x1 should be different from the value of x3). Finally, an objective function f is given. An *optimal solution* consists of an assignment of values to the xi such that: 1) all constraints in C are satisfied, i.e. the solution is valid, and 2) the assignment yields an optimal value for the objective function, f.

COPs typically exhibit a high degree of *epistasis:* the choices made during the search are closely coupled. In general, highly epistatic problems are characterised by the fact that no decomposition in independent subproblems is possible. In that case, it is difficult to combine subparts of two valid solutions into another valid solution. That is why the efficiency of genetic algorithms (GAs) - which typically search by combining features of different "solutions" - decreases significantly for higher degrees of epistasis.

The last couple of years a number of methods for constraint handling have been proposed within the GA community. The first one - genetic repair [Miihlenbein, 1992] - removes constraint violations in invalid solutions generated by the genetic operators, such as mutation and crossover. The second one uses decoders such that all possible representations give rise to a valid solution [Davis, 1988]. A third one uses penalty functions, see for example [Richardson *et al.,* 1989]. This approach defines the fitness function as the objective function one tries to optimize minus the penalty function which represents the "degree of invalidity" of a solution. All three approaches are however problem specific: for every COP one has to determine a good decoder, a good genetic repair method or a penalty function which balances between convergence towards suboptimal valid solutions (when the penalty function is too harsh) or towards invalid solutions (when too tolerant a penalty function is used).

A subclass of COPs, those involving only numerical linear (in)equalities, can be elegantly solved with GAs. In this case, the space of valid solutions is known to be convex. Michalewicz and Janikow [1991] used this property to define genetic operators which always generate valid solutions.

We tackle here the general class of COPs. The problem description of a COP typically contains constraints which implicitly describe which portions of the search space contain valid solutions. Here, we discuss how these constraints, which come with the problem specification anyway, can be used to improve the search efficiency of a GA by limiting the search space

it has to explore. Hence, the constraints provide cheap domain specific knowledge which can be used to augment domain independent search methods such as GAs. GSSS gets its generality from its embedding in the standard state-space search paradigm.

We first describe the general framework for the integration of constraint programming in GAs. Next, we discuss empirical results on a simple COP. Finally, we describe some extensions of our approach in the context of factory scheduling.

2. Constraint Programming for COPs.

Constraint programming has established itself as a good technique for solving COPs. The effective use of constraints during the search for solutions made it possible to solve "real-world" COPs, even when these exhibit a combinatorial nature.

A typical constraint program proceeds as follows. First, a *constraint network* is generated. This involves the creation of the variables xi, their domains Di, and the constraints between the variables. Next, the constraint based search algorithm repeats the following *selection-assignment-propagation cycle:* select a variable whose domain contains more than one element, select a value from the domain of that variable, assign the chosen value to the chosen variable (i.e. the variable's domain becomes a singleton containing this value)[1], finally propagation is performed. This propagation process executes all constraints defined on the reduced domain. This might further reduce the domain of other variables. For every domain which becomes reduced to a singleton the propagation process is recursively applied. The propagation algorithm above is known under the name *forward checking.*

The search algorithm described above can easily be described as a standard state-space search. With each search state a set of potential solutions can be associated. This set is simply the product of all domains, i.e. DIxD2x... x Dn. At each choice point (assignment), the domain of a variable is reduced to a singleton, followed by constraint propagation. This reduces the set of solutions associated with a state because the domains get smaller. Whenever a domain becomes empty no solution exists for the given choices. Or, in other words, the current state is a dead end. When finally all domains are reduced to a singleton a solution has been found.

The use of constraints allows for early detection of dead ends. Hence, the search procedure can skip branches of the search tree which do not lead to a valid solution. This may considerably reduce the amount of back-tracking. Another advantage of constraint programming is that one can state the constraints of the problem domain in a natural way. A good introduction to constraint programming can be found in [Van Hentenrijck, 1989].

[1] Here we use pure random variable and value selection. The use of more intelligent selection heuristics would obviously further improve the results presented in this paper.

3. A Genetic Representation for Search States

GSSS operates on search states instead of on complete solutions. Or, in other words, it operates on partial solutions instead of on complete solutions. For highly epistatic problems this is particularly advantageous because it is much easier to generate valid partial solutions than valid complete solutions to these problems. The same is true for the genetic operators: combining two valid complete solutions into another valid complete solution often turns out to be difficult. As we will see below, search states leave more flexibility for combining them in a "more complete" search state. We will also discuss how the quality (i.e. fitness) of a search state can be defined in terms of the quality of the potential solutions associated with the search state.

The representation of the search states on which GSSS operates is straightforward. It heavily relies on the fact that a search state is uniquely determined by the choices (i.e. value assignments) made to reach it from the initial search state. In the standard selection-assignment-propagation algorithm the number of choices is at most equal to the number of variables xi. Hence, a string of length n (the number of xi) is used. If at the search state to be represented the variable xi is not yet assigned a value, i.e. its domain is not a singleton, then the i-th element of the string consists of a ?. If, on the other hand, the search state contains the assignment (choice) xi=v then the i-th element is filled in with v. Consider, for example, the string ?15???2?. It represents a search state in which x2, x3, and x7 are assigned the values 1, 5, and 2, respectively. The representation used here was originally introduced by Hinton and Nowlan [1987] to study how individual learning can guide evolution. We use the term *PIG-representation* when referring to this Partially Instantiated Genotype representation.

4. The Algorithm

Our algorithm is based on GENITOR [Whitley, 1989], a well known genetic algorithm. After the generation of an initial population, GENITOR repeats the following steps: 1) select two parents from the population. This selection is biased towards individuals with a high rank in the population which is sorted on fitness; 2) a new individual is generated from these parents through the application of genetic operators (see below); 3) its fitness is calculated; 4) if this fitness is higher than the worst individual in the population then the child is inserted in the sorted population. At

the same time, the worst individual in the population is removed.

New, however, is the use of constraints in three components of the GA: during the creation of the initial population, during the fitness calculation, and during the application of the genetic operators. In these three modules GSSS uses both representations discussed above: the PIG-representation of a search state, and its corresponding constraint network. The individuals in the population are PIG-strings. The constraint network is used during the generation of the strings, during the application of genetic operators, and during the fitness calculation. Below, we describe how this is done.

4.1. The Creation of the Initial Population

GSSS starts from the two representations of the initial search state: 1) the PIG-string containing only ?s, and 2) a constraint network containing the variables, with their initial domains and the given constraints. Next, the *selection-assignment-propagation cycle* (see section 2) is executed an a priori determined number of times. Each cycle replaces a randomly chosen ? of the PIG-string with a value randomly chosen from the domain of the corresponding variable. Next, GSSS updates the constraint network: it reduces the domain of the chosen variable to the singleton containing the chosen value. Propagation ensures that the domains - from which the values are chosen - are consistent with the assignments made so far.

Interestingly, this generation of individuals is a constraint satisfaction problem itself. It is however simpler than the original problem because only a portion of the total number of choices need be made: the ?s represent choices which are left open. Because of this we are often able to insist on validity of the states in the initial population (with respect to the given constraints). Hence, no domain becomes empty as a result of the choices present in the individual. In our experiments, only a small amount of back-tracking was the price to be paid.

4.2. Fitness Calculation

We define the fitness of a search state as the value of the objective function for the best solution which can be reached from it through further assignments. Obviously, an exhaustive search through the set of potential solutions will often be far too expensive. Hence, we explore only an *h* priori determined number of "randomly chosen search paths". The constraint network corresponding with the search state to be evaluated is used as a starting point for each search path. Each path consists of a number of selection-assignment-propagation cycles. The random selection mechanism accounts for the random nature of the search process whereas the propagation enforces consistency between the choices. Search along a path stops when a) all domains are reduced to a singleton, i.e. a solution has been found, or b) propagation results in an empty domain, i.e. no valid solution exists for the choices made so far. We can effectively say that the searches explore the regions around a given search state. We define the fitness of a search state as the value of the objective function of the best solution found. Obviously, there is no guarantee that a valid solution will be found during the random searches starting at a given state. It is useful to give such states a (low) fitness. This fitness value should reflect the usefulness of the individual choices made to reach this state so that recombination can make use of valuable genetic material. We will see examples of this later.

The fewer ?s in a search state description the higher the probability that the searches find the same solutions. In the limit, every search starting from a fully instantiated search state - containing no ?s - always returns the same solution: itself. In order to provide selective pressure towards states containing fewer ?s, a secondary criterion is used to order states for which the best search yields the same quality. In that case, the state from which this solution quality is found most often is considered the fittest. Hence, this state is inserted before the other one in the sorted population.

Notice that the fitness value is a lower bound for the best solution which can be reached from the search state. It is important to stress here the usefulness of the propagation. It considerably limits the chances of ending up in a dead end. Hence, the chances of finding valid solutions are much higher than when no propagation were used. This way the promise of the state is estimated more reliably. The empirical comparisons later in this paper demonstrate this.

4.3. The Genetic Operators

Let us now look how crossover can be augmented through the use of constraints. As discussed above, there is no guarantee that combining choices of two "good" search states yields a search state from which a solution can be reached. GSSS uses constraint checking to remove inconsistencies introduced by crossover. Figure 1 illustrates this mechanism on a problem where the constraints require that all (eight) variables get a different value. In a first step, a (one-point) crossover operator generates the PIG-string, ?15???25. In this case, the randomly chosen crossover point is located after the fourth element. Obviously, the search state represented by this string cannot be expanded into a valid solution because of the two 5s. That is why a second step — constraint checking — is added. Starting from the initial constraint network the following process is repeated. First, one of the assignments in the PIG-string is randomly chosen. This assignment

is then done, i.e. the corresponding domain is reduced to a singleton. Next, propagation is activated. This process is repeated for all assignments in the string in random order. Each time a domain becomes empty the last assignment is discarded, i.e. the constraint network is restored to its state before that assignment. Furthermore, that last assignment is also deleted from the search state description. As a result, only a set of assignments mutually consistent with the constraints is retained. That is why, in our example, constraint checking only retains one 5 (see figure 1). Notice the generality of this approach: it is independent of the particular crossover used. As a matter of fact, it consists of a general genetic repair method for search states.

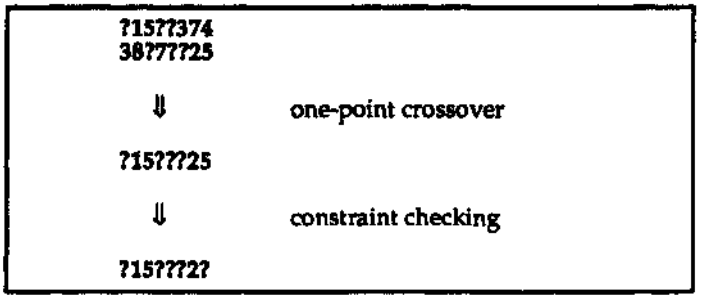

Figure 1: Our crossover operator extends the standard 1-point crossover with constraint checking.

Finally, mutation is applied to the result of crossover. The probability of mutating a PIG-string element is proportional to $(n - diff)/n_2$. Where n is the length of the string (i.e. the number of variables), and diff is the number of locations on the PIG-string which are different for both parents. Mutation changes a ? in an element randomly chosen from the domain of the corresponding variable. Values, on the other hand, are replaced by another randomly selected value from the domain or by a ?.

5. An Empirical Study

In this section, we introduce the bench-mark problem - an optimization variant of the well known n-queens problem — that allows us to illustrate our approach and to test it empirically. The n-queens problem consists of placing n queens on a nxn chess board so that no two queens attack each other (i.e. they are not in the same row, column, or diagonal). A typical constraint programming representation of this problem uses n variables xi. Each variable represents one column on the chess board. The assignment x2=3 indicates that a queen is positioned in the third row of the second column. Initially, the domains associated with the xi are instantiated to {1,2, ...,n). The constraints for this problem are simple:

- $xi \neq xj \qquad i \neq j$
- $|xi - xj| \neq |i-j| \qquad i \neq j$

The first line above prohibits two queens to be placed in the same row. The second ensures that no two queens are on a same diagonal. Note that the column constraint (only one queen is allowed per column) is implicit in the representation.

In order to turn the n-queens problem into an optimization problem we randomly assign a real number between 0 and 10 to each of the locations on the board. The COP-variant of the n-queens problem can now be described as follows: find a solution to the n-queens problem such that the sum of the values associated with the n locations where the queens are positioned is maximized.

We concentrate here on 30-queens COPs. The members of the initial population contain 15 assignments, and hence also 15 ?s. Furthermore, 25 searches are done to evaluate a state. The objective function for a valid solution is the sum of the values associated with the 30 locations where the queens are positioned. For invalid solutions this sum is divided by two. In the latter case, only the assigned variables contribute to the fitness. This way, the fitness value also yields information on sets of good locations even when none of the 25 searches find a valid solution.

In order to allow for empirical comparison we construct a "constraint-free" algorithm for the three components described above. For the generation of the initial population and the calculation of the fitness we simply drop the propagation from the selection-assignment-propagation cycle. This way the value-assignments in the initial individuals are randomly drawn from the initial domains, i.e. no consistency is enforced between the choices. Obviously, this algorithm often generates invalid search states which cannot be completed into valid solutions.

The constraint-free version of fitness calculation starts with the constraint network corresponding with the search state to be evaluated. The domains in this network reflect the assignments of the search state and the reductions resulting from propagation. During the 25 searches no further propagation is done, i.e. the domains are not changed anymore. Or, in other words, the domains - from which the values are chosen - do not reflect the choices made during the search.

The last constraint-free component involves the genetic operators. We used the same mutation operator as described above, only crossover is changed. It uses a conventional operator (here one-point) without constraint checking.

Up to now we introduced six algorithms. Let us call these i+, f+, o+, i-, f-, o-. The letters i, f, and o refer to the component, i.e. the creation of the initial population, the calculation of the fitness, and the application of the genetic operators. A minus refers to the variant

which uses no constraints, whereas a plus refers to the algorithm which does use constraints. A triplet consisting of one i, one f and one o algorithm represents a GA. We use i-f-o-, "the constraint-free GA", as a baseline for comparison.

Figure 2 depicts the results of our tests. The x-axis of this graph represents the number of offspring generated. The y-axis represents the average best-so-far fitness value, i.e. the fitness value of the best search state found so far averaged over 20 runs. All test runs use a population size of 200 and generate 5000 offspring. Performance data is collected every time fifty children have been generated.

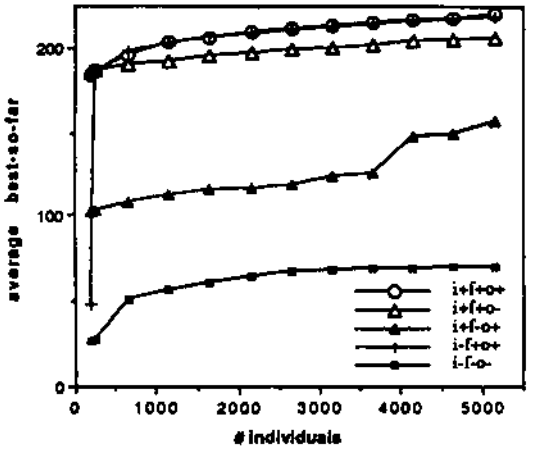

Figure 2: Empirical comparison of the five algorithms

A couple of interesting observations can be made from figure 2. The performance of i+f+o+ and i-f+o+ is virtually the same. Although the difference in the average value of the best element of the initial population is quite large (183.7 for i+f+o+, only 48.172 for i-f+o+) this difference has virtually disappeared after the creation of 50 children (184.45 versus 186.057). This because i- is unlikely to generate valid search states. Hence the low fitness value of the individuals in the initial population. The crossover of i-f+o+, however, ensures validity of the offspring with respect to the constraints. Hence, the performance of both algorithms quickly becomes similar. From this one can conclude that it is not worthwhile to insist on validity of the members in the initial population. These initial individuals only seem to act as a source of genetic diversity.

The performance of two other GAs (i+f-o+ and i+f+o-) demonstrate the pay-off of using constraints during the calculation of the fitness and the genetic operators, respectively. The average best-so-far of i+f-o+ after the creation of 5000 children is considerably lower than that of the initial population of i+f+o- and i+f+o+ (see figure 2). This because the random searches of f- are unlikely to find a valid - let alone an optimal - solution. Hence, i+f-o+ often considerably underestimates the fitness value of a search state. As a result many individuals from which good solutions can be reached may not get the chance to reproduce at all. Or, in other words, their genes might be lost. This lack of focus during the genetic search not only causes a low average quality of solution. It is also responsible for the large variation between the best solution quality found in different runs. The standard deviation of the best-so-far at the end of each of the 20 runs is 5.33 and 49.17 for i+f+o+ and i+f-o+, respectively (see table 1). This while their average best-so-far equals to 220.9 and 160.96, respectively! Some of the runs of i+f-o+ did not even find any valid solution during the generation of 5000 offspring. Table 1 also shows an average increase in solution quality of 7% when constraint checking is used during crossover (the average best so far of i+f+o+ and i+f+o- is 220.9 and 206.5, respectively.

	average best-so-far	standard deviation	worst	best
i+f+o+	220.9	5.33	208.42	234.42
i-f+o+	219.1	4.42	210.88	227.00
i+f+o-	206.5	6.77	197.15	225.10
i+f-o+	160.96	49.17	121.92	240.25
i-f-o-	69.9	20.23	54.87	112.36

Table 1: Performance data of twenty runs of the five GAs on the same 30-queens COP (pop-size=200, #offspring=5000)

6. Job Shop Scheduling

The general framework described in this paper grew out of earlier work on the use of constraint programming for scheduling [Paredis and Van Rij 1992]. Paredis [1992] describes an interesting modification to the general approach described here. It takes into account the volatile environment in which scheduling takes place: orders may be cancelled, machine breakdowns may occur etc. In such a volatile environment one should be able to reactively revise schedules in response to unexpected events. Instead of putting a large effort in finding one optimal schedule, we aim at finding states from which a number of different good schedules can be reached. Whenever one cannot stick to a given good schedule, then one can - to some extent at least - search locally around these states for another feasible schedule. In order to achieve this the fitness calculation takes into account not only the best solution found during the random searches but also the number of different solutions and the average quality of the solutions. By changing the relative importance of these features the search process explores other regions of the search space trading off the density and variation of solutions, the quality of the best solution, and the average quality of the solutions.

7. Discussion

In the previous sections we saw that the combination of constraint programming and GAs considerably enhances the solution quality. Here, we investigate the differences between our approach and standard GAs.

Holland showed that for problems with a moderate degree of epistasis, operators that splice together parts of two different individuals might yield good solutions. For such problems, sets of functionally dependent genes are relatively small. In that case, it becomes possible to use a string representation in which "correlated" genes are placed close to each other. Once the GA finds good values for these genes, they are unlikely to be split apart during sexual reproduction. Analogously, independent genes should be located far apart from each other on the string. Otherwise there will be too little exploration: suboptimal values for these genes are unlikely to become disrupted. This typically results in premature convergence. In general, a linear string might not be sufficient to place interacting bits close to each other, and to place non-interacting bits far apart. This is particularly the case when addressing problems with a high degree of epistasis.

Our approach is aimed at a large class of combinatorial problems which can be stated in terms of constraints. No search control knowledge is needed. The knowledge present in the problem specification is fully exploited through the constraints. We still use the standard linear representation. But now dependence need not necessarily be translated into physical proximity. The use of constraint propagation allows the GA to take into account long-distance interactions. The constraints represent the epistatic linkages.

There is an additional remark to be made here: the degree of interaction between a group of "genes" is not necessarily constant over the entire search space. Our algorithm operates on search states with different levels of instantiation. Hence, it searches at different hierarchical levels. At different levels, different dependencies may dominate. The opposite goes as well: the tightness of a dependency often depends on the assignments in the search state. The standard static linear representation is unlikely to be able to cater for these changing dependencies in an adequate way. Here again, the constraints explicitly enforce the relevant dependencies even when they span a wide distance in the genetic representation.

8. Conclusion.

GSSS combines the advantages of both: GAs and constraint programming. GAs have proven to be good search algorithms for large, moderately epistatic, problems. Hence, we use genetic search to explore the large search spaces of COPs. A more thorough exploration of the region around a search state is done through constraint-based search. Furthermore, domain knowledge in the form of constraints allows the GA to deal with higher degrees of epistasis.

We presented a genetic representation for search states of COPs which allows for the combination of both paradigms: genetic search and constraint programming. Next, we discussed the incorporation of constraints in three components of a GA. Our empirical study clearly shows the advantage of the use of constraints during the fitness calculation and during reproduction. Especially, the former plays a crucial role in obtaining an efficient hierarchical search.

References

[Davis, 1988] Davis, L., Applying Adaptive Algorithms to Epistatic Domains, Proc. IJCAI-88.

[Hinton and Nowlan, 1987] Hinton, G., Nowlan S.J., How Learning Can Guide Evolution, *Complex Syst.* 1,1987.

[Michalewicz and Janikow, 1991] Michalewicz, Z., and Janikow, C.Z., Handling Constraints in Genetic Algorithms, *Proc. Fourth International Conference on Genetic Algoritms,* Morgan Kaufmann, 1991.

[Miihlcnbein, 1992] Muhlenbein, H., Parallel Genetic Algorithms in Combinatorial Optimization, *Proc. Computer Science and Operations Research: New Developments in their Interfaces,* ORSA, Pergamon Press, 1992.

[Paredis, 1992] Paredis, J., Constraints as Background Knowledge for Genetic Algorithms: a Case-study for Scheduling, *Proc. Parallel Problem Solving from Nature,* Manner, R., Manderick, B., (eds.), Elsevier Science Publishers, 1991.

[Paredis and Van Rij, 1992] Paredis, J., and Van Rij, T., Simulation and Constraint Programming as Support Methodologies for Job Shop Scheduling , *Journal of Decision Systems,* Vol. 1, nr. 1, Pomerol, J.-C, (ed.). Editions Hermes, 1992.

[Richardson et al., 1989] Richardson, J.T. ; Palmer, M.R.; Liepins, G.; Hilliard M., Some Guidelines for Genetic Algorithms with Penalty Functions. *Proc. Third Int. Conf. on Genetic Algorithms,* Morgan Kaufmann, 1989.

[Van Hentenrijck, 1989] Van Hentenrijck, P., *Constraint Satisfaction in Logic Programming,* MIT Press.

[Whitley, 1989] Whitley, D., The Genitor Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trails is Best. *Proc. Third Int. Conf. on Genetic Algorithms,* Morgan Kaufmann, 1989.