Genetic Planning Using Variable Length Chromosomes

Alexandru Horia Brie*
Ecole Polytechnique, 91128 Palaiseau, France
Phone: +33 6 33 03 05 18
alexbrie@gmail.com

Philippe Morignot
AXLOG Ingenierie
19-21, rue du 8 Mai 1945, 94110 Arcueil, France
Phone: +33 1 41 24 31 19 Fax: +33 1 41 24 07 36
philippe.morignot@axlog.fr

Abstract

This paper describes a genetic planning system, i.e., a program capable of solving planning problems using evolutionary techniques. As opposed to other approaches in Genetic Planning, we use a variable length chromosomes model in addition to a complex fitness function and several enhancements of the Simple Genetic Algorithm (Holland 1975), such as multipopulations, population reset, weak memetism, tournament selection and elitist genetic operators. Our genetic planner is tested on standard planning domains and problems (described in PDDL), is used for parameter and performance analysis, and is compared to previous work. Results show efficiency in memory management and greater solving power than the predecessors'.

Introduction

Domain-independent planning is a fundamental and dynamic field of A.I. that has been tackled with a large amount of methods and algorithms, generating amazing advances during the last decade, among which: GraphPlan and its descendants, SATPLAN, HSP and its descendants (e.g., HSP version 2.0, FF), constraint-programming planners (e.g., CPT), mixed-integer programming planners, etc.

Among all these directions of planning research, a rapidly growing subfield is represented by Genetic Planning (*GP*), meaning the development of genetic algorithms for solving planning problems. Introduced in (Koza 1992), *GP* has been mostly developed during the last few years, with results from (Spector 1994), (Muslea 1997b), (Muslea 1997a), and lately with (Westerberg & Levine 2000), (Westerberg 2002). Such a recent interest in *GP* from the planning community might be surprising, since the main advantages promised by the *GP* approach in 1992 were reduced memory space required (smaller than that of other non-genetic planners) and higher complexity of problems that could theoretically be solved.

When taking this approach into consideration, we must first mention that very few papers have explored it (approx. 5): if genetic algorithms are well known, importing techniques from the evolutionnary field into the planning one still is in its first steps. So our first goal is to analyze the connection between evolutionnary techniques and A.I. planning, and see how it can help *reproducing* the best results of *GP* found so far. A second goal is to extend these results, by developping a genetic planner that can either reach better performances (even before comparing them with the fastest planners outside *GP*, e.g., FF) or be able to interpret a wider range of planning problems (i.e., integrate a a larger sub-set of PDDL). A third goal comes from the lack of practical implementation data required for planners' comparison within the *GP* subfield. So we attempt at providing as many implementation details as possible (within paper space limitation), in order for our genetic planner to help future dialog within this subfield.

 The authors thank Nelly Strady-Lecubin and anonymous reviewers for useful comments on drafts of this paper.

Planning

A planning problem is described as the process of finding a sequence of actions that best achieves some goals, given an initial situation and action templates. In order to do so, the agent needs a simplified model of the world, generally represented as a set of predicates representing possible states, and a set of actions that can act on these predicates. Some simplifying assumptions are: atomic time, deterministic effects, omniscience and sole cause of change (Weld 1999).

A widely used representation for planning problems is the STRIPS one, which declares each action as a list of preconditions (positive or negative facts that must be true before executing the action), an add-list (facts that the action makes true) and a delete-list (facts that the action makes false).

Evolutionary Algorithms

Evolutionary Algorithms are a class of general problem solving methods inspired by genetics and mainly used in optimization and combinatorial problems. The idea is to simulate the evolution of a population of individuals encoding approximate solutions of a given problem — an approach inspired by the Darwinian theory of the evolution of species.

An individual is rated by a *fitness function* (indicating how much adapted to its environment the individual is) and selected for reproduction proportional to these fitness values ("The fittest survives"). Reproduction consists in creating a new population composed of new individuals, from the individuals of the previous population (the "parents"), by means of (nature-inspired) so-called genetic *operators*,

hoping to transmit valuable features to their offspring. The choice of the parents is carried out by the *selection* operator. This evolution step is repeated, eventually finding optimal individuals that encode a solution of the problem, or "good enough" individuals, representing approximate, sub-optimal solutions.

Among all evolutionary algorithms, the most common are the Genetic Algorithms (GAs) (Holland 1975), whose individuals are linear sequences (called *chromosomes*) of independent elements (called *genes*). Evolution operators include *binary crossover* (i.e., switching sub-chromosomes between two parent individuals) and *unary mutation* (i.e., random change of a gene of a child). We are not aware of analytical results on GAs (e.g., convergence speed), except for very simple domains.

Genetic Programming, introduced in 1985 by N. L. Cramer and developed in (Koza 1992), simulates the evolution of programs and functions: Individuals are tree-like structures of source code instructions in a specific pseudo-language; the fitness value is computed by simulating the program's execution, and the genetic operators are specific, working with subtrees of the individual's *genome* (i.e., the set of gene-encoded information of the individual).

Connecting A.I. Planning to Evolutionary Algorithms

While most classical planning algorithms handle planning by traversing the search space in a tree-shaped path, advancing to neighboring states using heuristic functions, pruning inaccessible branches or backtracking whenever stuck into dead ends, evolutionary algorithms solve similar problems using a different approach — possibly incomplete and possibly leading to sub-optimal solutions, as it is the case for recent heuristic planners.

A large amount of individual "agents" (the *chromosomes*) are seeded into the search space, try to estimate the quality of the current points (distance from the solution) and are reseeded afterwards in the neighborhoods of the best found such positions. The entire population of agents jumps this way across states according to probabilistic mutation transformations and crossovers, selection or reproduction. Selection provides the agents with the possibility to traverse the space towards states of better quality. Mutation moves the agent into its own neighborhood. Crossover allows greater "leaps" by combining old positions of several agents into newer ones.

The efficiency and simplicity of the process relies on an uninformed search: the chromosomes are guided on the search space by means of probability distributions, unaware of the search space configuration. This increases the speed of the evolution process, allowing to use a large number of chromosomes with greater chances of success.

Our current research covers problems with totally ordered, totally instantiated plans that can be seen as sequences of actions, although other representations may exist. In our approach, *a gene encodes an action identifier and a chromosome encodes a plan*. This way, a search for a correct plan for a given problem in A.I. Planning becomes a search for the optimal chromosome of action identifiers in Evolutionary Algorithms.

As a consequence, this kind of search makes no reasoning about the plans, nor does it search for the next action to be added to a given subplan; just like in the case of other combinatorial problems, it bases its performances on the right choice of the fitness function, on the evolution model and on the genetic operators.

The Genetic Planner

Our genetic planner solves planning problems described in a sublanguage of PDDL that covers typed STRIPS domains. These domains comply with the usual simplifying assumptions of (Weld 1999), with additional restrictions raised because of the implementation: the constants are modeled by typed variables, there are no composite types and the plans obtained by evolution are totally ordered and instantiated.

A feature distinguishing it from the few evolutionary planners previously developed is the possibility to use strongly typed domains, generic variables and several few logical operators. Indeed, the work of (Spector 1994), (Koza 1992) or (Westerberg & Levine 2000) approaches only untyped planning domains, while (Muslea 1997b) allows typed domains in a specific syntax and representation, incompatible with STRIPS. However, the main conceptual difference is the use of a GA-like structure (with various improvements) and the variable length structure of the chromosomes (as opposed to fixed-length ones).

The behavior of our genetic planner complies with the standard Evolutionary Algorithms paradigm: (i) after creating an internal model of the problem, it creates the seed population by randomly instantiating individuals (i.e., random plans containing more or less bugs/flaws); (ii) the evolution process consists in selecting individuals, applying genetic operators to them and adding the resulting children to the population of the next generation; (iii) the fitness value of each individual is then updated and the evolution process is repeated until a final condition is met (e.g., a solution plan is found, or a time out is exceeded).

The paper is organized as follows: first, we describe the model; then we describe experiments on PDDL domains; then we compare our approach to previous work in this subfield; finally we sum up our contribution and indicate future research directions.

Model

Domain Representation

The problem model consists in an internal representation of the PDDL structures: types, objects, primitive facts, applicable actions, the problem's initial state and goals. For simplicity reasons, the object types are encoded as numbers with the default value *zero* being associated to the general type. The facts are represented by their identifiers and the types of their parameters. The actions are represented by their identifiers, the facts appearing in the preconditions or postconditions and a binding environment which connects the facts parameters with the real objects of the domain.

Following (Weld 1999), the initial state of the problem and its goals are modeled as two actions, *InitialAction* and *FinalAction*, the former without preconditions and the latter without postconditions.

Gene

Different encoding variants have been suggested by previous authors: (i) having numeric identifiers for all possible instantiations of actions on all parameters; (ii) mixing action identifiers with their parameters in a linear representation (Ruscio, Levine, & Kingston 2000) (the same gene could be interpreted either as an action identifier or as an action parameter, depending on the rest of the chromosome); (iii) assuming a maximum number of parameters per action, using fixed-length genes (Muslea 1997b) (in order to simplify typing, the object referenced by a parameter also depends on the action encoded in the same gene, through a rather complex mechanism).

Our planner uses a more direct approach instead: a gene is an atomic structure consisting of an action identifier and the list of its instantiated parameters (i.e., identifiers of the domain objects).

The length of this parameter list is given by the number of parameters required by each action, with the type correctness checked at the moment of their instantiation. At the creation of the gene, all parameters are completely instantiated to objects of the correct type (an additional mapping lattice from types to objects is used for that purpose).

In order to correctly deal with this particular gene structure, a special mutation operator is introduced (the *parameter mutation*, see subsection Mutation Operator below), which changes only a randomly chosen parameter in the gene's list of parameters. The parameter's new value is also checked for type correctness, as for the gene instantiation.

Chromosome

Since even for simple planning domains an upper limit for a plan length is unknown *a priori*, we believe that linear chromosomes of variable length are a more interesting and accurate model for our genetic planner — as opposed to arborescent genetic programming structures (Spector 1994), (Koza 1992) or fixed-length linear ones (Westerberg & Levine 2000), (Westerberg 2002). This choice leads to somewhat less intuitive genetic operators (see subsection Operators below), but seems closer to the classical planning philosophy.

The implementation class containing the chromosome also contains additional information (computed by the fitness function): the fitness value, the location of the first conflicting gene (i.e., the first action in the plan which cannot be executed), the location of the last conflicting gene.

Formally, a chromosome is an ordered list of pairs of action identifiers a_i, with $i \in [1, N]$, and bindings $(p_{i,j}, o_j)_{j \in [1, Param_i]}$, where $Param_i$ is the number of parameters of action identifier a_i and N is the length of the chromosome/plan. A gene is a pair of one such action identifier and its associated bindings. Each binding of action identifier a_i links a parameter $p_{i,j}$ of a fact of the action identifier a_i to an object indexed o_j. Thus, an entire chromosome C (or a plan of actions) is described as $C = (a_i, (p_{i,j}, o_j)_{j \in [1, Param_i]})_{i \in [1,N]}$. Note that once actions are known (at domain loading time), their parameters' identifiers $p_{i,j}$ are entirely determined.

As an example, let us encode the solution plan of the Sussman anomaly, given a unique $move(x, y, z)$ action (therefore indexed 1) that moves block x from block y on block z (respectively indexed 1, 2 and 3), and constants representing blocks A, B, C and $Table$ (respectively indexed 1 to 4) — note that we leave away the representation of the $Table$ staying clear (this may imply an additional action for putting a block on this $Table$), since our planner is not able to express conditions on action postconditions. The solution chromosome/plan, which is looked for by our planner (see section Experimental Results below), is $((1, ((1, 3), (2, 1), (3, 4)))$, $(1, ((1, 2), (2, 4), (3, 3)))$, $(1, (1, 1), (2, 4), (3, 2)))$ with the indices above. Note that all constant objects do not fit as values of fact parameters, because of object typing — a combinatorial shrink.

The Fitness Function

One of the most important components of our genetic planner, the fitness function, is a linear function combining various variables and user-defined weights. In a descending representation (common in genetic programming), a solution plan has the fitness 0, while the other chromosomes have a greater value. The fitness function parses the chromosome structure and simulates the execution of the underlying plan: (i) a *world state* containing the *InitialAction* of the problem is first set. (ii) all actions in the chromosome, if applicable, are executed in sequence by simulation. Testing the applicability of a subsequent action means verifying the validity of its preconditions in the state preceding this action. If all preconditions are satisfied in the current state, the postconditions of the add-list are added to the world state and those of the delete-list are removed from it. (iii) Final goal satisfaction is checked by trying to apply *FinalAction*, in a way similar to any other action in the plan.

Here are the main components of the fitness function:

- $number\_of\_conflicts$: counts the number of unsatisfied preconditions.
- $number\_of\_bad\_actions$: counts the number of actions which cannot be executed.
- $chrom\_size - first\_conflict\_position$: favors the chromosomes which begin correctly (means-driven plans).
- $chrom\_size$: favors long chromosomes up to a given threshold[1].
- $best\_sequence\_size$: favors chromosomes with longer correct subsequences, thus motivating creation of intermediate possible subplans that could serve as building blocks for subsequent solutions.
- $count\_collisions(FinalAction)$: favors plans that come close to the goals (goal-driven plans).

[1]This both prevents being stuck in local minima with small chromosomes, and does not have any effect on too long chromosomes.

All these components are weighted by configuration parameters described in a configuration file, which are individually set by the user in order to obtain the desired behavior (e.g., increase the performances).

Now the fitness function needs a way to execute bugged/flawed plans: If an action cannot be executed, it will be counted as such by the fitness function. Depending on a configuration parameter, two different approaches can be selected : (i) either ignore the bad actions; (ii) take them into account (by ignoring the preconditions and applying directly the postconditions anyway), as if they were executable. The reason for such a heuristic is to provide a less strict selection, allowing individuals to survive, although "bugged", in the hope that future mutations or crossovers might solve these chromosome problems.

Operators

As a starting point, our genetic planner uses common Genetic Programming operators (Westerberg & Levine 2000): crossover, reproduction and several kinds of mutation. In addition, with a given probability, an elitist selection is also used (Westerberg 2002) but extended to the mutation as well: in the new population, mutated or crossed-over offspring are used only if their fitness value is better than the one of the parents; otherwise the parents are used instead. This kind of elitism is a heuristic, the purpose of which is to avoid losing good individuals in exchange for their offspring. Since its success depends on the problem to be solved, it is applied with a probability parameterized by the user. This probability should be strictly below 1, otherwise the planner converges to local minima, losing the advantage of local search.

Crossover Operator This operator is one-point uniform: After having selected two parent chromosomes $C = (a_i, B_i)_{i\in[1,N]}$ (with B_i the bindings of C) and $C' = (a'_i, B'_i)_{i\in[1,N']}$ (with a similar meaning for B') with the selection scheme, a cutting point CP, less than N and N', is chosen (the same for both parents) using the cutting point heuristic (see below). Two offspring O and O' are created afterwards by joining the beginning of one parent to the end of the other — and vice versa. Formally, $O = ((a_1, B_1), \ldots, (a_{CP-1}, B_{CP-1}), (a'_{CP}, B'_{CP}), \ldots, (a'_{N'}, B'_{N'}))$ and $O' = ((a'_1, B'_1), \ldots, (a'_{CP-1}, B'_{CP-1}), (a_{CP}, B_{CP}), \ldots, (a_N, B_N))$.

Although we tried more complex schemes such as the two-cutting-points crossover (i.e., for each parent, two cutting points are used and the three segments resulting in each parent are interweaved) and the non-uniform crossover (i.e., different cutting points for the 2 parents), tests show that they decrease the performances when compared to the proposed crossover operator. The reason for this is based on the characteristics of the variable length chromosomes and on the plan representation: since the chromosome fitness depends on the number of conflict points (discontinuities), by attaching together randomly picked subchromosomes of the parents, we risk to bring together incompatible genes (the end of a subchromosome with the start of the other). Therefore, the number of conflicting points may increase if the number of subchromosomes increases too.

Therefore it is desirable to reduce the number of cutting points, or to split the chromosomes mostly at the conflicting genes positions, thus reducing the number of conflicting points introduced. This reasoning also applies to the mutation process.

Mutation Operators Here are the six operators we use:

- *Growth Mutation*: insert a random correct gene into a random position in the chromosome. Formally, this involves (i) randomly picking $G \leq N$, a value of the index i in a chromosome C, (ii) adding 1 to all indices $i \in [G, N]$ and to N, (iii) randomly picking G', a replacing value for G, and (iv) inserting $(a_{G'}, (p_{G',j}, o_j)_{j\in Param_{G'}})$ at location G in chromosome C. The values of the parameters (indices o_j) are chosen to be compatible with the type of each parameter.
- *Shrink Mutation*: erase a gene. Formally, this involves (i) randomly picking $G < N$, a value of the index i in a chromosome C, (ii) removing 1 from all indices $i \in [G+1, N]$ and from N. The first and last genes cannot be erased, since they contain the initial and final pseudo-actions identifiers.
- *Swap Mutation*: swap the position of two genes. Formally, this involves (i) randomly picking G_1 and G_2, $G_1 \neq G_2$ and both between 1 and N, and (ii) swapping the pairs $(a_{G_1}, (p_{G_1,j}, o_j)_{j\in[1,Param_{G_1}]})$ and $(a_{G_2}, (p_{G_2,j}, o_j)_{j\in[1,Param_{G_2}]})$.
- *Replace Mutation*: replace a gene by another random one with parameters of the correct types. Formally, this involves (i) randomly picking G, a value of index i in chromosome C and (ii) for the bindings in the G-th pair (action identifier, bindings), replacing all j of the terms o_j by different values j', such that each $o_{j'}$ is compatible with the type of $p_{i,j}$.
- *Parameter Mutation*: while still keeping the gene at its place, change the value of a random chosen parameter to another correctly typed one. Formally, this involves (i) randomly picking $G \leq N$, a value of index i in chromosome C, (ii) randomly picking j', a value of the index j in the G-th pair (action identifier, bindings) of chromosome C, and (iii) in the j'-th binding pair, replacing o_j by $o_{j'}$ — restricted to values of j' for which the type of $o_{j'}$ is compatible with the one of p_j.
- *Heuristic Mutations*: other heuristic operators have been tested, such as removal of some or all of the conflicting genes in a chromosome (formally, this is the same as the shrink mutation above, but for values of G given by the fitness function), or removal of duplicated instances of the same gene (formally, this still is the same as the shrink mutation above, but for a value of G given by a pre-computation). Great care should be taken since such operators may be harmful by reducing the population's diversity and possibly accelerating the convergence to a sub-optimal solution. On the other hand, applied rarely enough, they contribute to the genetic material quality.

The probability of occurrence of any of the mutation operators can be specified individually into a configuration file.

Selection

In the field of Evolutionary Algorithms, various selection schemes exist. While a popular one is the *roulette selection*[2] or the *stochastic sampling*[3] (Pohlheim 2004), the selection technique that we use is the *tournament selection* (Blickle & Thiele 1995), which preserves population diversity thus favoring the plan-finding process. It consists in randomly picking a fixed number of chromosomes (a user-defined parameter, usually ranging from 2 to 5) and selecting the chromosome which has the best fitness value. The number of chromosomes in the tournament directly influences the diversity rate and the convergence rate: the higher it is, the faster the convergence (assuming however the risk of premature convergence towards local minima).

The Cutting Point Heuristics

The chromosome's first and last conflicting positions (as discovered by the fitness function) are heuristically used in the selection of cutting points for mutation and crossover operators: with a high probability the chosen cutting point is a random one, while the first or last conflict points in the chromosome are chosen otherwise.

This heuristic greatly improves the building rate of almost correct chromosomes (which have few conflict points), but also reduces the population's diversity by possibly disfavoring interesting individuals; therefore, it should be applied with care as well, just like the other heuristics.

Seeding Heuristics

Another heuristic that we use is the weak memetism on seeding: Using a local search process (backtracking on the actions and objects), we detect all correct possible first actions to be applied from the initial state. Afterwards, all random chromosomes take as first action one of these possibly correct ones. This process is executed only once, before creating the population and is not time-consuming, since it is limited to one correct action only. This weak hybridation (memetism) of the genetic process, although still a heuristic only, proves to be helpful by reducing the number of generations.

Multipopulations (Islands, Ecological niches)

They represent a concept mainly used in Parallel Genetic Algorithms (Cant-Paz 2000): Several populations evolve in parallel and migration of few individuals occur with a given frequency (i.e., every n generation). The model greatly improves the performances by reducing the risk of premature convergence towards local minima. In our approach, the best individual (the *elite*) of an island is copied to the position of the worst individual of the neighboring island. The topology of the island set is circular (see Figure 1 for an example with 4 populations), although other topologies are possible.

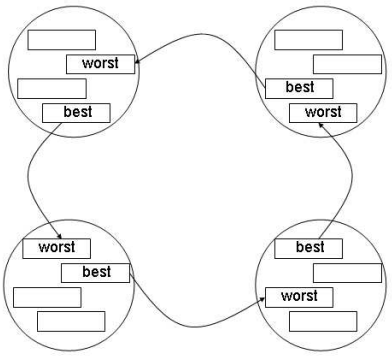

Figure 1: Circular multipopulation

Population Reset

Even with these precautions being taken, the evolution process still risks to get stuck in local optima. In a procedure similar to that used in *Micro Genetic Algorithms* (Coello Coello & Toscano Pulido 2001), if stagnation is detected, the population is reinitialized by replacing most of the chromosomes with random ones and keeping only some of the most interesting chromosomes, possibly slightly modified. Stagnation occurs when no improvement of the population's elite has been detected during a given number of generations.

Algorithm

Let TS be the tournament size, PC and PM be the crossover and mutation probabilities — all defined in a configuration file. Let $C_{p,q}$ denote the p-th chromosome of the q-th population (denoted P_q). Matrix $C_{p,q}$ has $Chroms$ rows and $Pops$ columns. Our genetic planner acts as follows:

1. a parser for the reduced sublanguage of PDDL converts the given problem into an internal representation - the *World Model*;
2. a parser for the configuration file creates the internal structures required by the genetic algorithms - the *Evolver*;
3. the population is initialized with random chromosomes of varying lengths[4] (the initial number of genes in a chromosome ranges between empirical limits depending on the problem);

[2] A selection operator in which the chance of a chromosome being selected is proportional to its fitness (or rank). This is where the concept of survival of the fittest comes into play.

[3] The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness, then $next\_population\_size$ equally spaced pointers are placed over the line selecting the corresponding chromosomes.

[4] The only purpose of the spread of the initial chromosomes' size is to accelerate the convergence. Our planner behaves similarly even if the initial chromosomes have a fixed length (e.g., 10 genes), because of the growth mutation — which occurs at a rate parametrized by a configuration file.

4. the evolution process, supervised by the *Time Manager*, occurs by using genetic operations such as crossover, mutation and selection for each q-th population:
   (a) select one or two chromosomes using tournament selection. Formally, this involves (i) randomly picking TS chromosomes from P_q; (ii) computing the fittest one $C_{p_f,q}$ — the one that minimizes the fitness function; (iii) repeat (i) and (ii) to exhibit a second fittest chromosome $C_{p'_f,q}$.
   (b) apply crossover and/or mutation on the selected chromosome(s). Formally, this involves (i) drawing a random uniform value V between 0 and 1; (ii) if $V \leq PC$, applying crossover on $C_{p_f,q}$ and $C_{p'_f,q}$, which exhibits 2 offspring $C'_{p_f,q}$ and $C'_{p'_f,q}$, and do nothing otherwise. (iii) drawing a random uniform value V' between 0 and 1; (iv) if $V' \leq PM$, applying mutation on $C_{p_f,q}$, which exhibits a mutated chromosome $C''_{p_f,q}$, and do nothing otherwise.
   (c) compute the offspring's fitness value by simulating the execution of the plan represented within it and by counting the flaws/bugs;
   (d) apply elitism selection if required (select either the offspring or its parent, according to their fitness value), with a given probability;
   (e) add the result(s) to the next population. Formally, this involves adding $C'_{p_f,q}$, $C'_{p'_f,q}$ and $C''_{p_f,q}$ to the next version, denoted P_q^{next}, of population P_q.
   (f) repeat from (a) until $size(P_q^{next}) = Chroms$, in which case P_q is replaced by P_q^{next}.
5. the *Time Manager* decides if migration or population reset should be made, and executes it accordingly. Formally, this involves (i) if the q-th population elite was not updated during the last `time_before_shaking_population` generations, resetting P_q; (ii) if `isolation_time` generations has elapsed, perform migration.
6. repeat from 4 (this counts the number of generations) until a solution is found, or a time out is exceeded;
7. decode the solution from the internal format to a PDDL-like one.

Experimental Results

The previous model is implemented[5] in the C++ language using the g++ compiler and the GNU tools Bison and Flex for the problem and settings files parsers.

The program was mostly tested on a Linux workstation with a 800 MHz Pentium III processor and 512 MB RAM. Because of the variable-length chromosomes, the processes of fitness evaluation and copying chromosomes, and therefore the entire evolution process, have uneven time lengths. The time required for generating a new population depends linearly on the number of populations and the number of chromosomes within it, but depends non-linearly on their uneven sizes.

As a benchmark measure, the number of generations needed to find a solution is used, since it is an indicator often used in Evolutionary Algorithms literature, and also since the load of the workstation's processor can highly vary, which obviously has a large impact on the machine's performances and on the resolution time. However, on the given computer, at reasonable processor load, for a 1500-chromosome population and a mean chromosome size of 15-20 genes, the duration of a generation ranges between 0.5 and 2.0 seconds in real time.

Domains and Problems

In order to assess the program's performances, we use test problems from various domains: the *blocks world*, the modified blocks world (Spector 1994) and the typed domain of *Gripper*.

Blocks World Its representation contains four types of actions and five types of facts (predicates):

```
(:predicates (on ?x ?y) (on-table ?x) (clear ?x) (arm-empty)
   (holding ?x))
(pick-up ?ob1)
(put-down ?ob)
(stack ?sob ?sunderob)
(unstack ?sob ?sunderob)
```

We test the resolution of the following three problems:

- P1: the Sussman anomaly. Although this example is often used to demonstrate the effect of inter-dependency among subgoals, we use it to illustrate the inherent performance gap between informed search of classical planners and uninformed search of GAs.
- P4: initial state: { A/C/B/table }; final state: { B/A/C/table }
- P5: initial state: { B/C/A/D/table }; final state: { D/A/C/B/table }

Spector's Blocks World In order to be able to compare our results with those of (Spector 1994), we used his particular blocks world domain, which only contains two types of facts and two actions:

```
(:predicates (on ?x - block ?y) (clear ?x))
(newtower ?table - table ?x - block ?y)
(puton ?x - block ?y ?z)
```

The problem that we test on this specific domain still is the Sussman anomaly.

Gripper Gripper is an example of typed domain. STRIPS typing can be modeled by using specific facts corresponding to each type, but a dedicated syntax also exists in the PDDL/STRIPS version. We test the same problem using both representations of the domain, i.e. typed Gripper (*GrpT*) and untyped Gripper (*GrpUT*). The domain consists of a robot having two arms *grippers*, moving *ball* objects from one room to another. The three types described are *ball*, *gripper*, *room*, and the possible actions are *pick(?object, ?room, ?gripper)*, *move(?from, ?to)*, *drop(?object, ?room, ?gripper)*.

[5] Source code available upon request to the authors.

The problem solved involves 5 rooms (roomA to roomE), 4 balls (ball1 to ball4), and has the initial state: { robot in roomA, ball1 in roomA, ball4 in roomA, ball2 in roomB, ball3 in roomC } and the final state: { ball1 in roomC, ball4 in roomB, ball2 in roomA, ball3 in roomB }.

Methodology

Our planner includes a fairly large amount of parameters to set (e.g., weights of the fitness function, probabilities). To sort their importance, we test the program's behavior for the above problems by modifying several of these parameters, one at the time, while keeping all the others unchanged, i.e., set to default values. The non-constant parameter varies on a given range, thus giving the possibility to draw conclusions on optimal values.

Although this approach is not optimal because many of the program's parameters are mutually interdependent, we find it nonetheless interesting, by revealing useful clues on the default values to use.

For each problem and parameter, the tests are reproduced several times, in order to observe an average behavior — because of the stochastic nature of genetic algorithms, results and numbers of generations may highly vary from test to test, even for the same problem and parameter set.

After this parameter analysis, we use the best parameter set found so far to check the performances in real-time of our planner on a specific domain.

Population Size

It is natural to consider the generic performance of a GA as proportional to the number of individuals. However, since a larger population requires longer time per generation, an optimum combination is looked for. Figures 2 and 3 show the results (number of generations required) for the problems above, obtained by varying the number of chromosomes in each one of four populations.

The tests show that the best results appear with population sizes of about 300 chromosomes per population. Although increasing the number of individuals makes the number of generations required by the solution slightly decrease, the time required by each generation is increased instead. Therefore an optimal amount of about 1200-1500 individuals, equally distributed among a reduced number of populations, provides the best performances.

Multipopulation Number

This parameter selects the number of populations (islands). If set to 1, the advantages of multipopulation disappear, the behavior being the one of a classical GA. While it is expected to notice an increase in performance with the number of populations, we also must account for the execution time, just like in the previous test. The performances are shown in Figures 4 and 5.

Similarly to the previous test, a general improvement is noticeable with the increase of the number of populations. A minimum point is at 4 multipopulations, proving that too many populations are not suitable either. Since the time required by a generation also increases with the number of

Size	P1	P4	P5	Spector	GrpUT	GrpT
150	12	23	137	17	221	11
150	26	11	101	17	353	13
150	6	12	27	6	1003	11
200	8	12	28	12	273	12
200	62	54	79	12	50	16
200	6	35	61	18	81	15
250	12	17	67	8	51	12
250	21	38	44	8	53	10
250	15	23	192	6	44	16
300	14	30	37	4	23	4
300	15	17	21	9	172	9
300	15	9	29	14	51	11
350	12	11	23	7	73	16
350	14	13	83	6	11	12
350	11	15	36	8	259	10
400	5	8	35	10	44	14
400	7	10	46	6	141	10
400	7	6	49	9	55	7

Figure 2: Population size results (generation numbers/problem)

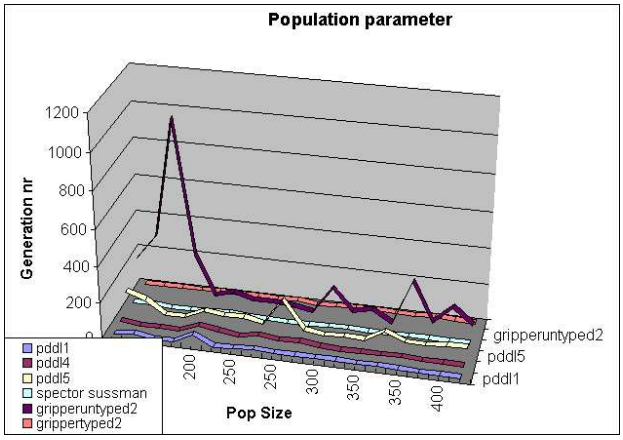

Figure 3: Population size graph

Size	P1	P4	P5	Spector	GrpUT	GrpT
2	94	21	64	17	108	18
2	19	39	47	12	228	14
2	14	121	151	22	131	18
3	10	13	23	8	166	18
3	41	18	94	43	65	17
3	21	11	48	9	301	10
4	9	47	21	9	31	6
4	21	24	30	10	101	12
4	10	23	34	6	47	16
5	11	20	30	8	75	16
5	13	21	24	10	35	4
5	22	39	116	17	36	14
6	16	144	52	14	46	14
6	10	8	25	5	39	13
6	13	15	81	7	105	13

Figure 4: Number of populations results

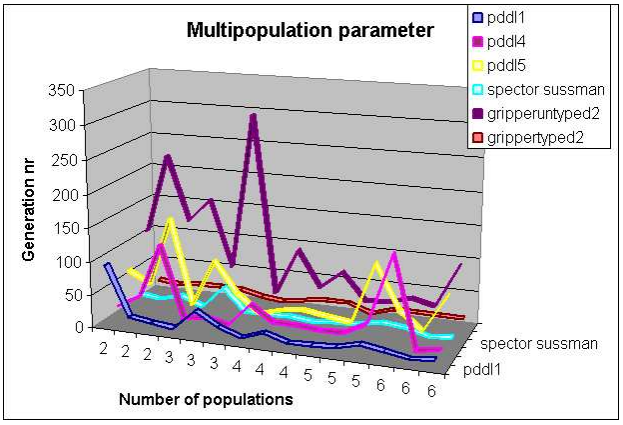

Figure 5: Number of populations graph

Size	P1	P4	P5	Spector	GrpUT	GrpT
2	16	24	27	6	48	16
2	27	19	54	14	191	23
2	21	28	37	14	129	16
3	14	70	33	13	39	6
3	20	16	26	10	191	9
3	12	67	50	4	51	9
4	11	31	49	16	203	8
4	2	107	19	8	72	9
4	15	15	39	44	87	10
5	12	156	45	7	92	11
5	10	19	44	8	160	6
5	4	10	29	6	54	9

Figure 6: Tournament size results (for tournament selection)

chromosomes, the conclusion is that the optimum performance is provided by a group of 3 or 4 populations.

Tournament Size

Tournament size plays an important role in the selection process: By augmenting its size we can speed up the convergence with the risk of premature convergence towards local minima. Several tests are run in order to look for the optimum value, as shown in Figures 6 and 7.

In this case, the results vary highly, as seen in the charts. For some types of problems (simpler linear ones, with less local optimums: P1, GrpUT, GrpT), stricter selection process increases the performances, while decreasing them for more difficult problems (P4, P5, Spector).

The best trade off for tournament size value between convergence speed and population diversity seems to be situated between 3 and 4, thus justifying our intuitive choice of 3 as a default value.

Scaling Up

To observe how the approach scales up, we measure the performances of our planner on the typed Gripper domain (see Figure 8) with a parameter set close to the best one found above. The problems grow in difficulty and require permut-

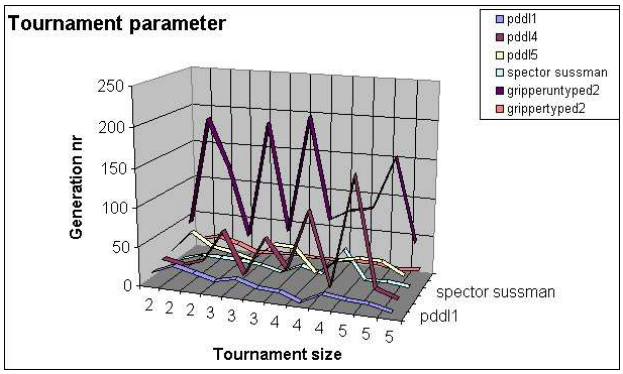

Figure 7: Tournament size graph

ing a variable number of balls into a corresponding number of rooms. Missing vertical bars indicate that a given time out (set to 1000 generations) has been exceeded — the best chromosomes obtained in such cases contain minor flaws or partially fulfill the goals. Problems above size 5 exceeded this generation limit and therefore are not shown. Once again, each test problem is run several times (run #1 to #5), in order to attempt at coping with the stochastic nature of our approach. Given the average real time taken by our planner for computing one generation (1,25 seconds), the average performance on the average run can be deduced. Finally, here are the parameter values used for these runs of the various instances of gripper, and their brief explanations:

```
isolation_time  7 // #generation before migration among islands
rand_gene_break_random  0.8 // probability that the mutation /
  crossover point is random, instead of heuristically chosen
crossover_prob  0.8 // crossover probability
mutate_act_replace_prob  0.07 // probability of replace mutation
mutate_param_replace_prob  0.06 // prob. of parameter mutation
mutate_swap_prob  0.05 // probability of swap mutation
mutate_add_prob  0.07 // probability of growth mutation
mutate_erase_prob  0.06 // probability of shrink mutation
mutate_elim_duplicate  0.03 // probability of duplicate removal
mutate_maturation  0.001 // prob. of conflicting genes removal
probability_of_consecutive_mutations  0.2 // probability of
  mutating several consecutive genes
elitist_mutate  0.6 // probability of using elitism for mutation
elitist_cross  0.7 // probability of using elitism for crossover
cross_and_mutate  0.5 // probability of mutating the offspring of
  a crossover
force_update_even_if_collision  1 // flag to ignore the bad
  actions in the plan's fitness
// Eight weights for components of the fitness function
conflict_pound  0.05 // For the conflicts
conflicting_actions_pound  0.01 // For the conflicting actions
purpose_distance_pound  0.3 // For the unsolved goals
repeating_actions_pound  0.1 // For duplicated actions
conflict_position_pound  0.02 // For the first conflict pos.
longestsequencepond  -0.01 // For the longest correct subplan
number_of_actions_pound  -0.001 // For the chromosome size
upper_nactions_limit  10 // threashold below which
   number_of_actions_pound is valid
time_before_shaking_population  15 // #generation with elite
  stagnation before reset
shaking_population_total  130 // max value for picking odds relatives
shaking_population_random_chroms  127 // relative odds for adding
  a randomly seeded chromosome
big_mutate_number_of_mutations  5 // #mutation to be applied on
  the chosen chromosome
shaking_population_best_sol_mutations  1 // relative odds of
  mutating a previous chromosome
shaking_population_elite_mutations  1 // relative odds of
  mutating the population's elite
population_shake  1 // flag for population reset
```

These results show that the number of generations of the average run grows exponentially as a function of the data

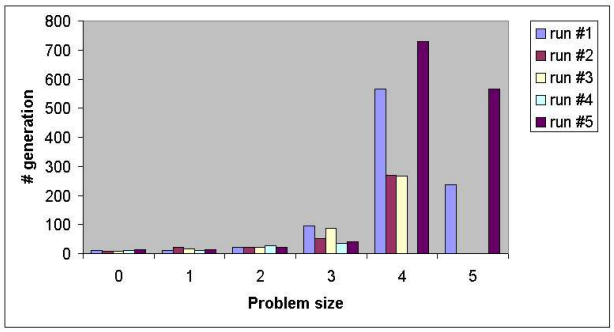

Figure 8: Performances on the domain Gripper

size (i.e., the number of balls and arms) involved in the problem.

Related Work

Very few papers concerning domain-independent planning by evolutionary techniques exist. The first ones use Genetic Programming, usually implemented in the interpreted LISP programming language, which allows a great flexibility and ease of programming, but also might involve a speed bottleneck. We mention here the three that we are aware of: John Koza's dedicated pseudo-planners (Koza 1992), Lee Spector's Genetic Planner (Spector 1994) and Ion Muslea's *Synergy* (Muslea 1997b) (Muslea 1997a). All of them use genetic programming operators and data structures as well as custom-designed fitness functions simulating the execution of the plan encoded by the chromosome and returning a value that combines the number of conflicting actions, the length of the chromosome and a few other features. Planners such as Koza's and Synergy use non-standard descriptions of planning domains and operators, in order to directly encode LISP functions inside. This approach has huge expressive power because computing the fitness value actually executes various LISP programs, but also might slow down the fitness evaluation process — depending on the complexity of these functions.

Were we to compare our planner with Koza's, Spector's or Muslea's, we notice some important differences: Koza's dedicated planners makes use of huge populations (about half a million individuals) and of extremely powerful parallel machines — which none of us has access to. On the other hand, Muslea and Spector use medium-sized populations of about 200 arborescent chromosomes, with interesting results in less than 100 generations, though the computing time for each generation is probably important, since complex operations are being executed. For several of Spector's test examples, the performance of our program is better or comparable (regarding the number of generations), for medium difficulty problems and first solution search. However, other examples and best-solution tests[6] show better results (still regarding the number of generations) for Spector's planner.

[6]Continuing the evolution even after finding a first solution, in order to try to optimize it. Our program could also search for best solutions with minor adjustments.

A different approach is the one of GenPlan (Westerberg & Levine 2000), inspired by more recent works in GAs. This one makes use of a linear encoding of the chromosome as a fixed-length array of genes, large populations, tournament selection, elitist crossover and a simple fitness function counting *good* genes and *active* genes in the chromosome[7]. Their tests show impressive performances, sometimes outperforming the planner BlackBox on some particular tests with large problems.

About performances, comparing the results of our genetic planner to those of previous researchers in this field is a very hard task, because of lack of relevant data (e.g., duration of generations, hardware characteristics, implementation details). In order to use a common indicator, we analyze the number of generations needed (which has lots of drawbacks) for finding the solutions. Therefore we attempt at providing as many details as possible, which can be used in further GP research for more accurate comparisons.

Another important point is that the other researchers use only carefully-chosen problems, illustrated by just a *single* execution. We believe that this is an inaccurate method, since the execution time may vary greatly because of the random operators used. So we take care of using mean values over a range of runs, to cope with this random aspect.

Another aspect is that, in special cases, despite our efforts to avoid it, the planner gets stuck in local optima and seems to loop indefinitely. Because of the mutation operator, chances are that it might eventually get out of these local optima and continue the solving process. Still, in a Monte Carlo vision of the process, after any amount of time, the program can be stopped in order to decode a possible "good enough" pseudo-solution, either partial or false, but in both cases the best solution so far (an anytime view over GAs). This problem occurs in other genetic planners as well (Westerberg & Levine 2000), but this issue, although disturbing *a priori*, is not that important, since we may always interrupt the evolution and restart a new solving process.

Conclusion and Future Work

In this paper, we presented a genetic planner using variable-length chromosomes, instead of fixed-length ones, and including various genetic schemes (complex fitness function, multipopulation, population reset, weak memetism, tournament selection and elitist genetic operators). The reason is that, in order to be efficient, the fixed-length chromosome approach needs to use large chromosome sizes, with only few genes actually modeling a plan, the others being inactive. They therefore need more memory and more powerful machines to run on. Even so, the fixed-length approach implies a limit of the plan size, which assumes a theoretical incapability of solving problems that require longer plans.

[7]In their approach, the chromosomes length is much bigger than the one of the plans contained in them, the rest of the chromosome being filled with dummy (inactive) genes — in a way inspired by real (human) genetics. An *active* gene is a gene contained in the subchromosome encoding the plan, while a *good* gene is an active gene that was also correct, i.e. not conflicting with the rest of the plan.

Our variable-length approach has none of these drawbacks: the memory requirements are minimal and plans can potentially grow to any needed length.

Due to the high number of parameters to set (which is common in GAs), we ran tests to find the best value of some of them: population size (around 300 individuals), number of population in a multipopulation scheme (between 3 and 4) and tournament size value (between 3 and 4). We also use typed domains for the STRIPS subset of PDDL, and use an elitist mutation as well. We also present pure performance results.

Our future research directions include:

- Obviously, genetic planners (including ours) cannot compete yet with, for example, heuristic planners such as FF[8]. However, the advantages an evolutionary technique can add to a planner are huge, especially for large or very complex problems. The best approach of the matter is probably the hybrid one: classical planning could provide a partial subplan to serve as starting points for genetic evolution (Westerberg & Levine 2001), reaching better memetism (see subsection Seeding Heuristics above); otherwise, classical planners can be used as genetic operators, adding some informed search features to the evolutionary process[9]; conversely, genetic algorithms can be used to obtain starting points for classical planning processes, thus greatly reducing the complexity of the search space.
- Even without hybridation, importing recent advances from the field of GAs can enormously improve GP, for example concerning the structure of the chromosomes and the machinery of the evolutionary processes, from using fixed-length chromosomes or tree-shaped chromosomes like in Genetic Programming, to using new revolutionary Genetic Algorithms, such as *Fast Messy Genetic Algorithm* (Goldberg *et al.* 1993) or *Linkage Learning Genetic Algorithm* (Harik & Goldberg 1997).
- Finally, PDDL is a complex language, and the STRIPS sublanguage covers only a part of it. Extended modeling capacity can be added to our planner, by allowing the use of universal quantifyers, conditional effects, domain axioms, logical or probabilistic operators that could eventually make more use of the stochastic inherent structure of genetic algorithms. Great care should be taken, though, since solving more expressive problems implies using highly complex fitness functions (meaning plan interpreters that can deal with bugged/flawed plans and give them quality estimations).

References

Blickle, T., and Thiele, L. 1995. A comparison of selection schemes used in genetic algorithms. Technical Report 11, Gloriastrasse 35, 8092 Zurich, Switzerland.

Cant-Paz, E. 2000. *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers.

Coello Coello, C. A., and Toscano Pulido, G. 2001. A Micro-Genetic Algorithm for Multiobjective Optimization. In Zitzler, E.; Deb, K.; Thiele, L.; Coello, C. A. C.; and Corne, D., eds., *First International Conference on Evolutionary Multi-Criterion Optimization*. Springer-Verlag. Lecture Notes in Computer Science No. 1993. 126–140.

Goldberg, D. E.; Deb, K.; Kargupta, H.; and Harik, G. 1993. Rapidaccurate optimization of difficult problems using fast messy genetic algorithms. In *Proceedings of the 5th International Conference on Genetic Algorithms*, 56–64. Morgan Kaufmann Publishers Inc.

Harik, G. R., and Goldberg, D. E. 1997. Learning linkage. In Belew, R. K., and Vose, M. D., eds., *Foundations of Genetic Algorithms 4*. San Francisco, CA: Morgan Kaufmann. 247–262.

Holland, J. 1975. *Adaptation in Natural and Artificial Systems*. Cambridge, MA, USA: MIT Press.

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press.

Muslea, I. 1997a. A general-purpose AI planning system based on the genetic programming paradigm. In Koza, J. R., ed., *Late Breaking Papers at the 1997 Genetic Programming Conference*, 157–164. Stanford University, CA, USA: Stanford Bookstore.

Muslea, I. 1997b. SINERGY: A linear planner based on genetic programming. In Steel, S., and Alami, R., eds., *Fourth European Conference on Planning*, volume 1348 of *Lecture notes in artificial intelligence*. Toulouse, France: Springer-Verlag.

Pohlheim, H. 2004. *GEATbx: Genetic and Evolutionary Algorithm Toolbox for use with MATLAB Documentation*.

Ruscio, L.; Levine, J.; and Kingston, J. K. 2000. Applying genetic algorithms to hierarchical task network planning. In *Proceedings of the 19th Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG 2000)*.

Spector, L. 1994. Genetic programming and AI planning systems. In *Proceedings of Twelfth National Conference on Artificial Intelligence*, 1329–1334. Seattle, Washington, USA: AAAI Press/MIT Press.

Weld, D. S. 1999. Recent advances in ai planning. *AI Magazine* (2).

Westerberg, C. H., and Levine, J. 2000. Genplan: Combining genetic programming and planning. In *Proceedings of the 19th Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG 2000)*.

Westerberg, C. H., and Levine, J. 2001. Investigation of different seeding strategies in a genetic planner. In Springer., ed., *Proceedings of the 2nd European Workshop on Scheduling and Timetabling (EvoSTIM 2001)*.

Westerberg, C. H. 2002. Elite crossover in genetic planning. In Luke, S.; Ryan, C.; and O'Reilly, U.-M., eds., *Graduate Student Workshop*, 311–314. New York: AAAI.

[8] That is why we did not register to the ICAPS Planning Competition this year.

[9] In a similar way, the HSP planner uses Graphplan without delete lists as a heuristics for search.