An Evolutionary Metaheuristic Based on State Decomposition for Domain-Independent Satisficing Planning

Jacques Bibaï[1,2] **Pierre Savéant**[1] **Marc Schoenauer**[2] **Vincent Vidal**[3]
[1]Thales Research & Technology
Palaiseau, France
first.last@thalesgroup.com
[2]Projet TAO, INRIA Saclay & LRI
Université Paris Sud, Orsay, France
first.last@inria.fr
[3]ONERA – DCSD
Toulouse, France
Vincent.Vidal@onera.fr

Abstract

DAE$_X$ is a metaheuristic designed to improve the plan quality and the scalability of an encapsulated planning system. DAE$_X$ is based on a state decomposition strategy, driven by an evolutionary algorithm, which benefits from the use of a classical planning heuristic to maintain an ordering of atoms within the individuals. The proof of concept is achieved by embedding the domain-independent satisficing YAHSP planner and using the critical path h^1 heuristic. Experiments with the resulting algorithm are performed on a selection of IPC benchmarks from classical, cost-based and temporal domains. Under the experimental conditions of the IPC, and in particular with a universal parameter setting common to all domains, DAE$_{\text{YAHSP}}$ is compared to the best planner for each type of domain. Results show that DAE$_{\text{YAHSP}}$ performs very well both on coverage and quality metrics. It is particularly noticeable that DAE$_X$ improves a lot on plan quality when compared to YAHSP, which is known to provide largely suboptimal solutions, making it competitive with state-of-the-art planners. This article gives a full account of the algorithm, reports on the experiments and provides some insights on the algorithm behavior.

Introduction

Recent advances in the design of PDDL planners have focused on plan quality rather than on speed needed to obtain a single solution of eventually poor quality, as witnessed by the 6th International Planning Competition. Planners were given a fixed amount of running time, and their scores were based, for each benchmark domain, on their coverage (number of solved problems) and on the quality of their solutions with respect to various plan metrics. We think that this is an important step towards the design of planning systems able to tackle real-world problems, for which plan quality is generally a fundamental requirement. Another way to ensure solution quality is of course the use of optimal planners, but the size of the problems they can handle is by far lower than that solved by satisficing planners.

In that perspective, we propose DAE$_X$, a metaheuristic aimed at (i) guiding an encapsulated planner towards a solution of good quality, and (ii) increasing the scalability of that planner when facing difficult planning problems. The key components of DAE$_X$ are:

1- A *decomposition principle* used to divide a complex planning task into (hopefully) easier subtasks. We chose a state-based decomposition strategy: a planning task is sliced into a sequence of intermediate states that must be reached in turn before satisfying the goal. For reasons discussed later, these intermediate states are partial states only that are considered as subgoals during search. This decomposition principle relies on classical reachability planning heuristics. The idea of decomposing a search space in this way is not new (Korf 1987; Sebastia, Onaindia, and Marza 2006), but as we have very minimalistic information to compute such a decomposition, we consider this problem as an optimization problem and use a specialized optimization algorithm to try to discover the best decompositions.

2- An *encapsulated satisficing planner* used to solve each subtask. In principle any PDDL planner could suit, provided that it has a predictable behavior when applied to identical subtasks in order to ensure the convergence of the optimization process; even a stochastic planner such as LPG (Gerevini, Saetti, and Serina 2003b) could be used, by controlling its randomization seed. We chose for this purpose the YAHSP planner (Vidal 2004), which is extremely fast on many benchmark domains but suffers from poor solution quality and scalability problems in some domains. We consider particularly challenging the use of such a planner: will the proposed metaheuristic be able to improve both its scalability and plan quality?

3- An *optimization algorithm* used to drive the underlying planner towards a solution of good quality, by controlling it through the state-based decomposition process. We chose an evolutionary algorithm to conduct the optimization process, as these algorithms are known to have been very successful for many optimization problems, and to ensure a high diversification in the exploration of the search space. Indeed, a planner such as YAHSP often sinks into unpromising subtrees, either leading to dead-ends or bad solutions, without being able to visit better parts of the search space. The net effect of the optimization algorithm will be to force the planner to diversify the way it travels through the search space, and concentrate it simultaneously to several different promising parts.

DAE$_X$ builds on previous ideas implemented in DAE1 (Schoenauer, Savéant, and Vidal 2006; 2007) and DAE2

(Bibai et al. 2008), but differs from these works in several fundamental ways. Firstly, the decomposition principle of DAE1 was based on manipulations at the planning objects level, building intermediate states by combining predicate and constant symbols in a completely blind way. DAE2 introduced intermediate state computation at the atom level, but still in a blind way. DAE_X benefits from a time-based atom choice method relying on standard planning reachability heuristics (Haslum and Geffner 2000) and pairwise mutual exclusions between atoms. Secondly, DAE1 and DAE2 were based on the assumption that the best results should be obtained with an optimal planner such as CPT (Vidal and Geffner 2006). The resulting planners were effectively able to find very good solutions, but the cost of running YAHSP instead of CPT for each subtask is so much lower that DAE_X clearly explores vast parts of the search space that were out of reach for DAE1 and DAE2, making it able to outperform them both in scalability and quality. While DAE2 exhibited poor performance at IPC-6 — although the plan quality for the problems it could solve was often very good, if not the best (Bibai, Schoenauer, and Savéant 2009) — DAE_X is competitive with state-of-the-art planners in both coverage and quality, as demonstrated in the experimental section.

Divide-and-Evolve

This section presents the details of the basic implementation of DAE_X. As advocated in (Sebastia, Onaindia, and Marza 2006), the first ingredient for state decomposition is a decomposition principle. Previous works have tackled this issue by relying on theoretical bases, e.g., partitioning planning problems into subproblems by parallel decomposition (Chen, Hsu, and Wah 2006). On the opposite, DAE_X addresses the problem of finding a decomposition of a planning task $P = \langle A, O, I, G \rangle$ by turning it into an optimization problem: search for a sequence $S = (s_i)_{i \in [0,n+1]}$ such that the plan σ obtained by compressing subplans σ_i found by an embedded planner as solutions of $P_i = \langle A, O, s_i, s_{i+1} \rangle_{i \in [0,n]}$ has the best possible quality. Several crucial issues need to be addressed from the optimization point of view: identify the search space, define an objective function, and choose an optimization algorithm. The three issues are of course related: choosing a powerful method with proven convergence usually implies heavy restrictions on the search space and the objective function, and the practitioner then has to twist the problem at hand to make the chosen method applicable. The opposite route was chosen in DAE_X: avoid unnecessary restrictions on the search space or the objective function, and use an optimization algorithm that is both flexible and powerful enough to be able to tackle the resulting optimization problem.

Evolutionary Algorithms

Evolutionary Algorithms (EAs) are general purpose optimization algorithms that have been demonstrated to be highly flexible, but nevertheless robust, in handling such challenging optimization problems. EAs are metaheuristics based on a metaphor of the Darwinian evolution of biological populations (Eiben and Smith 2003): the interaction of *natural selection* (fitter individuals, with respect to the environment, survive and reproduce more than others) and *blind variations* (the genetic material is randomly modified when passed on from the parents to their offspring during reproduction) results in the *emergence* of individuals that are adapted to their environment. In the Artificial Evolution framework, individuals are candidate solutions of the optimization problem at hand, the environment is given by the value of the objective function, also called here *fitness*, selection amounts to choosing individuals with a bias towards good values of the fitness, whereas variation operators are stochastic moves in the search space that have to balance between *exploitation* of the previous good individuals, locally searching around them, and *exploration* of the search space, by creating new individuals far from already explored regions of the search space.

Note that selection procedures are problem-independent. Hence, implementing an evolutionary algorithm for a new problem requires to define the search space (or, equivalently, the *representation* of candidate solutions), the fitness function, and the variation operators, that are usually categorized into *mutation operators*, that modify a single *parent* to generate one *offspring*, and *crossover operators*, involving two or more parents to generate one or more offspring.

Representation for State Decomposition

In DAE_X, an individual is a state decomposition for the planning task at hand, i.e., a variable length list of states. However, searching the space of complete states would rapidly result in a combinatorial explosion of the size of the search space. Moreover, goals of a planning task are generally defined as partial states. It thus seems more practical to search only ordered sequences of partial states, and to limit the choice of possible atoms used to describe such partial states. However, this raises the issue of the **choice of the atoms** to be used to represent individuals, among all possible atoms.

Some results of previous experiments on different domains of temporal planning tasks from the IPC benchmark series (Bibai, Schoenauer, and Savéant 2009) have demonstrated the need for a very careful choice of the atoms that are used to build the partial states. This lead us to propose a new method to build the partial states, based on the earliest time from which an atom can appear. Such time can be estimated by any admissible heuristic function, e.g., h^1, h^2... (Haslum and Geffner 2000). The start times given by the chosen heuristic are used to restrict the candidate atoms for each partial state when building a sequence of partial states: a partial state is built at randomly chosen timestamps by randomly choosing among several atoms that can possibly appear at this time (this will be detailed more formally later). The sequence of states is hence built by preserving the estimated chronology between atoms (**time consistency**). The heuristic function h^1 has been used for all experiments presented here.

Nevertheless, even when restricted to specific choices of atoms, the random sampling can lead to inconsistent partial states, because some sets of atoms can be *mutually exclu-*

sive[1] (mutex in short). Whereas it could be possible to allow mutex atoms in the partial states generated by DAE$_X$, and to let evolution discard them, it is more efficient to a priori forbid them as much as possible. In practice, it is difficult to decide if several atoms are mutex. Nevertheless, binary mutexes can be approximated (i.e., not all pairs of mutually exclusive atoms can be discovered) with a variation of the h^2 heuristic function (Haslum and Geffner 2000) in order to build quasi pairwise-mutex-free states (i.e., states where no pair of atoms are mutex).

Last, but not least, the useful decompositions are those for which all resulting subproblems are easier to solve than the initial problem for the planner at hand. We use a purely syntactic (asymmetric) metric *dist* to evaluate the remaining difficulty of solving the current planning task: for any complete state i and partial state g, *dist*(i, g) is the number of atoms in g that are not in i. Other metrics could be envisaged, such as information given by a reachability heuristic, but the metric we used proved to be informative enough.

An individual of DAE$_X$ is thus represented as a variable length list of time-ordered partial states, where each state is a variable length list of atoms that are not known to be pairwise mutex. In the following, $T(a)$ denotes the estimated earliest starting time of a given atom a, $T = \{T(a) \neq 0 | a \in A\}$ the set of all such starting times, $\Delta(s) = max_{a \in s} T(a)$ the estimated earliest starting time of a given state s. For any atom a, $M(a)$ denotes the set of atoms which are mutex with a, according to the approximation based on the h^2 heuristic function. $\mathbb{U}$ denotes a uniform random draw from the set given as argument.

Fitness Computation

When addressing the planning task $P = \langle A, O, I, G \rangle$, the fitness of a state decomposition $S = (s_i)_{i \in [0,n+1]}$ (with $s_0 = I$ and $s_{n+1} = G$) is computed by calling an embedded planner to successively solve planning tasks $P = \langle A, O, s_i, s_{i+1} \rangle$. But two different situations should be distinguished here, depending on whether the embedded planner fails on one of the subproblems (the decomposition is then termed *infeasible*), or not. In both cases, there must be some fitness gradient, towards feasibility for infeasible individuals, and towards optimal plan quality for feasible ones.

The pseudo-code for the computation of the fitness is given in Algorithm 1. The main loop (lines 3-12) processes the intermediate states sequentially by calling the embedded planner on the corresponding planning subproblems (line 5). The initial state is the current state i, computed by actually running the solution plan of the previous subproblem (line 11); indeed, remember that g is only a partial state, whereas an initial state has to be complete. The goal is the currently processed partial state g. The last argument $b_{\max}$ is a boundary that is planner-dependent: its aim is to restrict the exploration, in order to discard subproblems that are too difficult (ideally, that are more difficult than the original global problem). Indeed, because there can be no guarantee on the difficulty of the subproblems, it is mandatory to restrain the embedded planner (it could also be a time boundary).

In the current implementation, the embedded planner YAHSP is constrained with a maximal number of nodes that it is allowed to expand for solving any of the subproblems. The actual boundary has been determined by a two-steps process: first, while evaluating the initial population, a very large number of nodes is allowed (e.g. 100000); the boundary is then chosen as the median of the actual number of nodes that have been used whenever a solution has been found during these evaluations of the initial population.

The embedded planner returns sol_k, the solution of the current subproblem, and the number of search steps (nodes, in the case of YAHSP) b_{done} that was needed to find it — unless it fails within the boundary $b_{\max}$ and returns a failure.

Algorithm 1 evaluate(Ind, planner)

```
Require: $I, G, b_{\max}, l_{\max}$
 1: $k \leftarrow 0$ ; $u \leftarrow 0$ ; $B \leftarrow 0$
 2: $i \leftarrow$ I ; $g \leftarrow \{\}$
 3: while $g \neq G$ do
 4:     $g \leftarrow$ nextGoal(Ind)
 5:     $(sol_k, b_{\text{done}}) \leftarrow$ planner.Solve$(i, g, b_{\max})$
 6:     if $sol_k = \perp$ then
 7:         return $(\perp, 10 \cdot k \cdot dist(i, G) + \text{length(Ind)} - u)$
 8:     else if length$(sol_k) > 0$ then //          avoid empty plan
 9:         $u \leftarrow u + 1$                          // useful states counter
10:         $B \leftarrow B + b_{\text{done}}$                     // total search steps
11:     $i \leftarrow$ ExecPlan$(i, sol_k)$                   // next initial state
12:     $k \leftarrow k + 1$                      // intermediate goal counter
13: $(Sol, Q) \leftarrow$ Compress$((sol_j)_{0 \leq j \leq k})$
14: return $(Sol, Q + \frac{length(Ind) - u + 1}{Q} + \frac{B}{l_{\max} \cdot b_{\max}})$
```

In the latter case, the fitness is set according to line 7: it aims at minimizing the syntactic distance *dist*(i, G) between the current initial state i and the final goal, that is also the last current complete state that has been reached. However, because the syntactic distance is by no way an accurate indicator of the actual remaining difficulty, the fitness also takes into account the number u of *useful* intermediate states, i.e., those intermediate states that require a non empty plan to be reached (line 8).

When the individual is feasible (all subproblems are solved by the embedded planner), a compression routine is used to compress all subplans (line 13), and the fitness is basically the total quality Q of the resulting global plan. This compression is made with a standard polynomial deordering procedure (Bäckström 1998) for temporal planning, or a simple plan concatenation for sequential planning. However, as in the infeasible case, it was necessary to penalize the individual by the amount of *useless* intermediate states, in order to avoid unnecessary bloat. Furthermore, a second additional term favors "easy" subproblems by penalizing all problems with the cumulated number of search steps B actually used by the embedded planner, divided by the product of the longest sequence of states allowed $l_{\max}$ and the boundary $b_{\max}$, leading to the formula of line 14.

The comparison between any two individuals assumes that a feasible individual is always preferred to an infeasible one, regardless of any fitness value. Two feasible individ-

[1]A set of atoms is a permanent mutex when there does not exist any plan that, when applied to the initial state, yields a state containing them all.

uals are compared according to the value returned line 14 while two infeasible individuals are compared according to the value returned on line 7.

Variation Operators

Variation operators modify the individuals in order to explore the search space. On the one hand, these operators should ensure the *ergodicity* of the search: any point of the search space must be reachable with a non-zero probability from any other point using a finite number of applications of variation operators. On the other hand, small modifications should be favored otherwise the evolutionary process is close to a random walk.

Algorithm 2 crossover(Ind$_1$,Ind$_2$)

```
1: $s_a \leftarrow \mathbb{U}(\text{Ind}_1)$                          // $\text{Ind}_1 = (s_i)_{1 \le i \le n}$
2: $t_b \leftarrow \mathbb{U}(\text{Ind}_2)$                          // $\text{Ind}_2 = (t_i)_{1 \le i \le m}$
3: if $\Delta(t_b) > \Delta(s_a)$ then return $(s_1, \ldots, s_a, t_b, \ldots, t_m)$
4:                       else return $(t_1, \ldots, t_b, s_a, \ldots, s_n)$
```

The **crossover** operator, as described in Algorithm 2, is the basic 1-point crossover for variable length representations: in order to recombine $(s_i)_{1 \le i \le n}$ and $(t_i)_{1 \le i \le m}$, it uniformly chooses some states s_a and t_b, and crosses the parts of both lists that maintain the chronology between atoms in a sequence of states, obtaining one offspring.

Algorithm 3 addGoal(Ind)

```
Require: $r$                                          // neighborhood radius
 1: $j \leftarrow \mathbb{U}([1, \min(\text{length(Ind)},\text{lastReached(Ind)})])$
 2: $s \leftarrow \{\}$                          // insert $s$ between $s_j$ and $s_{j+1}$
 3: $t \leftarrow \mathbb{U}(\{t \in T \mid \Delta(s_j) < t \le \Delta(s_{j+1})\})$
 4: $A_t \leftarrow \{a \in A \mid T(a) \in \text{neighbourhood}(t, r)\}$
 5: $A_m \leftarrow \{\}$                         // set of non pairwise mutex atoms
 6: while $A_t \neq \{\}$ do
 7:     $a \leftarrow \mathbb{U}(A_t)$
 8:     $A_m \leftarrow A_m \cup \{a\}$
 9:     $A_t \leftarrow A_t \setminus (\{a\} \cup M(a))$
10: $N \leftarrow \mathbb{U}([1, \#A_m])$                                // goal length
11: repeat
12:     $a \leftarrow \mathbb{U}(A_m)$             // choose uniformly an atom in $A_m$
13:     $s \leftarrow s \cup \{a\}$                                    // add to $s$
14:     $A_m \leftarrow A_m \setminus \{a\}$                        // remove from $A_m$
15: until $\#s = N$
16: insert(Ind, $s$, $j$)                               // insert $s$ after goal j
17: return Ind
```

Four different **mutation** operators have been used. Assume parent is $(s_1, \ldots, s_{\text{lastReached}}, \ldots, s_n)$, where $s_{\text{lastReached}}$ is the last state reached by the embedded planner ($s_{\text{lastReached}} = s_{n+1} = G$ if the individual is feasible, i.e., if all sub-problems have been solved). At the individual level, mutation **addGoal** randomly adds a state after state $j \le \min(n, \text{lastReached})$ as described in Algorithm 3: this new intermediate state may contain several atoms of A_t and several atoms of its neighborhood of radius r, where t is a time between $\Delta(s_j)$ and $\Delta(s_{j+1})$, and the neighborhood of radius r is the set of $2 \times r + 1$ immediate times before and after t including t. Reciprocally, mutation **delGoal** removes a state s_i, with i uniformly chosen in $[1, \min(n, \text{lastReached} + 1)]$. At the state level, mutation **addAtom** changes or adds (or both) one random atom in each state s_i ($i \in [1, \min(n, \text{lastReached} + 1)]$) as described in Algorithm 4, and mutation **delAtom** removes one uniformly chosen atom from state s_i, with i uniformly chosen in $[1, \min(n, \text{lastReached} + 1)]$.

Algorithm 4 addAtom(Ind)

```
Require: $p_c, p_a$  // relative probabilities to change or add an atom
1: for all $k \in$ [1,min(length(Ind),lastReached(Ind)+1)] do
2:     if $\mathbb{U}([0,1]) < \frac{p_c}{\text{length(Ind)}}$ then //              atom change
3:         $a \leftarrow \mathbb{U}(\text{Ind}[k])$
4:         $b \leftarrow \mathbb{U}(\{b \in M(a) \mid T(b) = \Delta(\text{Ind}[k]) \wedge \nexists c \in$
               $(\text{Ind}[k] \setminus \{a\}), b \in M(c)\})$
5:         $\text{Ind}[k] \leftarrow (\text{Ind}[k] \setminus \{a\}) \cup \{b\}$
6:     if $\mathbb{U}([0,1]) < p_a$ then //                             atom addition
7:         $a \leftarrow \mathbb{U}(\{b \in A \mid T(b) = \Delta(\text{Ind}[k]) \wedge \nexists c \in$
               $\text{Ind}[k], b \in M(c)\})$
8:         $\text{Ind}[k] \leftarrow \text{Ind}[k] \cup \{a\}$
9: return Ind
```

Initialization of the Population

The pseudo-code for the initialization is given in Algorithm 5. First, the number of states is uniformly drawn between 1 and the number of estimated earliest possible start times (algorithm 6 line 6); for every chosen time, the number of atoms per state is uniformly chosen between 1 and the number of atoms of the corresponding restriction (line 11). Atoms are then chosen one by one, uniformly in the allowed set of atoms, and added to the individual if not `mutex` with any other atom already there (lines 12 to 16).

Algorithm 5 generateIndividual(N)

```
Require: $T$                                     // candidate start times
 1: $D \leftarrow \{\}$                           // ordered list of timestamps
 2: repeat
 3:     $t \leftarrow \mathbb{U}(T)$
 4:     $T \leftarrow T \setminus \{t\}$
 5:     Insert($t$, $D$)                            // maintain $D$ ordered
 6: until $\#D = N$
 7: Ind $\leftarrow \{\}$                        // start building the individual
 8: for $t \in D$ do
 9:     $s \leftarrow \{\}$               // start building the intermediate goal
10:     $A_t \leftarrow \{a \in A \mid T(a) = t\}$  // atoms that can appear at $t$
11:     $n \leftarrow \mathbb{U}([1, \#A_t])$                       // number of atoms
12:     while $n \neq 0 \wedge A_t \neq \{\}$ do
13:         $a \leftarrow \mathbb{U}(A_t)$       // choose uniformly an atom in $A_t$
14:         $s \leftarrow s \cup \{a\}$                                // add to $s$
15:         $A_t \leftarrow A_t \setminus (\{a\} \cup M(a))$       // remove all mutex
16:         $n \leftarrow n - 1$
17:     Ind $\leftarrow$ Ind $+ \{s\}$         // add the new intermediate goal
18: return Ind
```

Evolutionary Loop

The first step of Algorithm 6 is the computation of the earliest start time for each atom $a \in A$ estimated with the given heuristic. The set T which gathers all potential start times will be used later in a mutation operator. The initial population is then set up by simply repeating calling the GenerateIndividual function up to the desired size. Then comes the main evolution loop (line 8). The offspring set is populated

Algorithm 6 DAEX(popSize, OffSpringSize, MaxGen, MaxChgt, p_{cross}, p_{mut}, $w_{addGoal}$, $w_{delGoal}$, $w_{addAtom}$, $w_{delAtom}$, b_{max}, l_{max}, r, p_c, p_a)

```
Require: planner, $h$    // embedded planner and heuristic function
 1: for all $a \in A$ do
 2:     $T(a) \leftarrow h(a)$                    // compute earliest start time
 3: $T \leftarrow \{T(a) \neq 0 \mid a \in A\}$              // candidate start times set
 4: pop $\leftarrow \{\}$                          // start building the population
 5: repeat
 6:     pop $\leftarrow$ pop $\cup$ {GenerateIndividual($\mathbb{U}([1, \#T])$)}
 7: until #pop = popSize
 8: repeat
 9:     offspring $\leftarrow \{\}$
10:     repeat
11:         Ind$_1 \leftarrow \mathbb{U}$(pop)
12:         if $\mathbb{U}([0,1]) < p_{cross}$ then
13:             Ind$_2 \leftarrow \mathbb{U}$(pop)
14:             Newind $\leftarrow$ crossover(Ind$_1$,Ind$_2$)
15:         else
16:             Newind $\leftarrow$ Ind$_1$
17:         if $\mathbb{U}([0,1]) < p_{mut}$ then
18:             $f \leftarrow \mathbb{U}_{weighted}$(addGoal, addAtom, delGoal, delAtom,
                    $w_{addGoal}, w_{delGoal}, w_{addAtom}, w_{delAtom}$)
19:             Newind $\leftarrow$ APPLY($f$, Newind)
20:         offspring $\leftarrow$ offspring $\cup$ {Newind}
21:     until #offspring = OffSpringSize
22:     for all Ind $\in$ pop $\cup$ offspring do
23:         Evaluate(Ind, planner)
24:     pop $\leftarrow$ SurvivalSelection(pop $\cup$ offspring)
25: until #gen > MaxGen OR noImprovementSince(MaxChgt)
26: return Evaluate(pop.BestIndividual, planner).Sol
```

with individuals from the population of the previous generation either as is or as the result of a crossover between two individuals and/or as the result of a mutation. The mutation is chosen non-uniformally (according to a weight) among four operators. All individuals are then evaluated before being submitted to the survival selection, which selects the population of the next generation from the parents+offspring (line 24) — see Section DAE$_{\text{YAHSP}}$ Settings for the actual implementation. The evolution stops either after a maximum number of generations or when no improvement has been observed since a given number of generations. Lastly one of the best individuals is evaluated to produce the best solution found.

Experimental Results

DAE$_{\text{X}}$[2] has been implemented within the Evolving Objects framework[3], an open source, template-based, ANSI C++ evolutionary computation library. Experiments have been conducted in order to assess the behavior of DAE$_{\text{X}}$ over different kinds of planning tasks: classical planning tasks, cost-based planning (actions with costs), and simple temporal planning tasks (actions with duration). IPC benchmarks domains have been used, from the corresponding IPC tracks. In order to select test domains, we have chosen for temporal planning tasks and planning with costs, all IPC-6 domains that can be tackled by YAHSP and several other domains from previous IPC competitions for which we have reference values[4]. For STRIPS problems, test domains were chosen according to their complexity as defined by (Helmert 2008), with the goal of having different types of complexity. The complete list of domains is given with the results in Table 1: altogether, 736 problems have been tested.

Furthermore, the results of DAE$_{\text{YAHSP}}$ have been compared with those of the best state of the art planners: LAMA (Richter, Helmert, and Westphal 2008), updated version, LPG (Gerevini, Saetti, and Serina 2003a; 2003b), and TFD (Eyerich, Mattmüller, and Röger 2009), updated version which, according to the authors, outperforms all state-of-the-art temporal planning systems, plus of course the embedded planner itself YAHSP (Vidal 2004).

Performance Measures

Experiments were done using a 2 GHz computer with a 6 MB cache and a 16 GB RAM, running Linux. All algorithms are given at most 30 minutes of CPU time for each run on each problem instance. Their **coverage** is then measured by the number of instances solved in each domain. The quality of the plans are evaluated using IPC rules. For a given instance i, let Q_i^* be the reference plan quality. The quality ratio for each planner is defined by Q_i^*/Q_i. The **quality score** of a planner for domain $\mathcal{D}$ is the sum over all instances of $\mathcal{D}$ of the quality ratios of this planner. The planner with the highest quality score is designated as the best performer on the domain. Note that if a planner cannot find a plan for a given instance after 30 minutes, its quality ratio is set to 0 for this instance.

However, DAE$_{\text{YAHSP}}$ and LPG are stochastic algorithms, and no firm conclusion can be drawn from a single run. Hence 11 independent runs have been performed on each instance in order to assess their robustness. Their **coverage** per domain is defined as the total number of instances that have been solved at least once. The **average coverage** of LPG and DAE$_{\text{YAHSP}}$ for a given domain $\mathcal{D}$ is defined as $\frac{\sum_{i;n_i>0} n_i}{\sum_{i;n_i>0} 1}$, where n_i is the number of successful runs (i.e., that found a plan) for instance i of $\mathcal{D}$. The average coverage hence lies in $[0, 11]$, the higher the better. Finally, the **average quality** of LPG and DAE$_{\text{YAHSP}}$ for domain $\mathcal{D}$ is defined as the sum over all solved instances i of $\mathcal{D}$ of $\frac{1}{n_i}\sum_{\{\text{run } j \text{ solved } i\}} \frac{Q_i^*}{q_j}$ where q_j is the quality of the plan found by run j — the closer to full coverage, the better.

DaE$_{\text{YAHSP}}$ Settings

One identified weakness of EAs is the difficulty in tuning their numerous parameters, as there exists no theoretical guidelines to help the practitioner. Users generally rely on their previous experience on similar problems, or use standard and expensive statistical methods, e.g., Design of Experiments (DOE) and Analysis of Variance (ANOVA). Experimental statistical procedures have been proposed (e.g., Racing (Yuan and Gallagher 2004)), that build on standard

[2]DAE$_{\text{YAHSP}}$ will be soon available under CeCILL-C license

[3]http://eodev.sourceforge.net/

[4]Reference values are either the best results of all IPCs, or the best values obtained with CPT (resp. DAE1, DAE2).

DOE and use the specificities of the Evolutionary Computation domain to reduce the amount of computations.

In order to tune DAE$_X$, (Bibai et al. 2009) proposed a two steps learning approach which involves choosing the probability and weights of each of the variation operators with Racing, and then choosing which predicates will be used to describe the intermediate goals with statistical analysis. In this paper, only the first step of (Bibai et al. 2009) approach has been used, over several domains of IPC benchmarks. The best parameter set output by the Racing procedure has be chosen as the common parameter configuration for all experiments of this paper, and is described below.

However, the Racing procedure (Yuan and Gallagher 2004) was limited to the parameters of the variation operators, and the **evolution engine** had been fixed according to preliminary experiments: population size is set to 100 and offspring size to 700, each parent generates exactly 7 offspring using variation operators. The survival selection is a comparison-based *deterministic tournament* of size 5: 5 individuals are uniformly chosen in the set of 800 parents+offspring, and the best of those 5 is chosen to become a parent of the next generation. Furthermore, the same stopping criterion has also been used for all experiments: after a minimum number of 10 generations, evolution is stopped if no improvement of the best fitness in the population is made during 50 generations, with a maximum of 1000 generations altogether. Finally, the parameters of the variation operators, as determined by the initial Racing phase, are the following: the probabilities of individual-level application of crossover and mutation (p_{cross} and p_{mut}) are (0.2, 0.8) and the relative weights of the 4 mutation operators (w_{addGoal}, w_{delGoal}, w_{addAtom}, w_{delAtom}) are (3,1,1,1). The neighborhood radius was set to 2, the longest sequence of states allowed l_{max} was set to $2 \times \#T$, and the relative probabilities to change or add an atom (p_c and p_a) were set to (0.8, 0.5).

Results

First column (resp. second column) of Table 1 shows for all algorithms the best coverage S$_{\text{planner}}$ (resp. quality Q$_{\text{planner}}$), together with the average quality of LPG and DAE$_{\text{YAHSP}}$, and the average coverage of DAE$_{\text{YAHSP}}$ (the average coverage of LPG is always equal to 11 and is therefore not presented). Last column is the ratio Q$_{\text{planner}}$/S$_{\text{planner}}$. The mean values of those figures across test domains are also provided, by domain category, and over all domains.

Figure 1 displays boxplots for the average number of states and atoms per state for the best decompositions obtained by DAE$_{\text{YAHSP}}$ on `zeno simple time` (the situation is similar on other domains). It shows that DAE$_{\text{YAHSP}}$ builds larger decompositions with more atoms per state as instances get harder — even though the settings are the same for all instances. DAE$_{\text{YAHSP}}$ thus seems to somehow grasp instance difficulty. Figure 2 shows two typical examples of the fitness behavior along evolution on `crew planning 30` and `openstacks simple time 30`. It highlights the learning power of evolutionary computation for an unknown problem structure, that seem very different between these two instances.

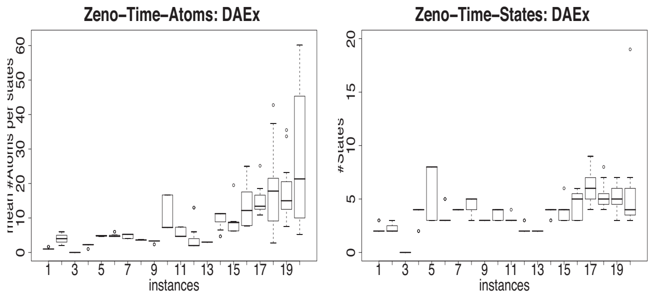

Figure 1: DAE$_{\text{YAHSP}}$ Diversity on `zeno simple time`.

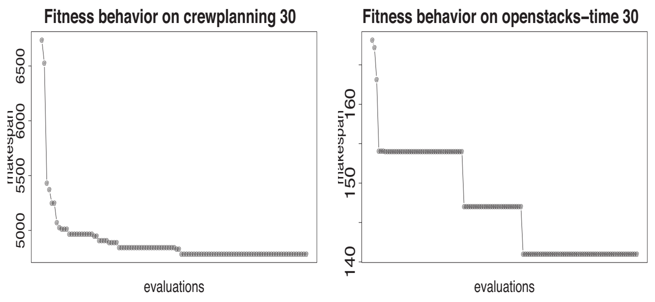

Figure 2: Fitness behavior of DAE$_{\text{YAHSP}}$ on `crew planning 30` and `openstacks simple time 30`.

Discussion

The first clear result is that DAE$_{\text{YAHSP}}$ solves significantly more problems (92.53% of total) than YAHSP alone (88.86%), much more (91.25%) than LPG (82.50%) and TFD (75.83%) on simple temporal planning, much more (94.14%) than LPG (79.69%) on STRIPS planning, and a little more than LAMA on STRIPS planning and cost-based planning. Then, DAE$_{\text{YAHSP}}$ has the best quality score (see last lines of Table 1) for all kinds of planning tasks. Furthermore, DAE$_{\text{YAHSP}}$ more often finds (see Table 1) either the reference value (which may be optimal), or a value greater than 90% of the reference value, and always finds a better plan quality than YAHSP alone (Table 1). Note that the best improvement on quality obtained by DAE$_{\text{YAHSP}}$ over YAHSP alone is on cost-based planning.

However, although the DAE$_{\text{YAHSP}}$ planner has the best quality score over all tested domains (last lines of Table 1), LPG has the best ratio on STRIPS domains and simple temporal domains (last lines of the corresponding sub-tables of Table 1). We believe that this is due to the use of a common parameter configuration for all experiments, and further work will investigate instance-specific parameter tuning. Nevertheless, there does not exist any absolute best method: even in the case where DAE$_{\text{YAHSP}}$ (respectively LPG) obtains the best ratio value on a given type of problems, there is always at least one domain of this type where the other planner performs better on all instances it could solve (see table 1). See for instance, the `pegsolitaire` (DAE$_{\text{YAHSP}}$ planner) domain for temporal planning tasks, and `elevator` (LAMA planner) and `pathways` (DAE$_{\text{YAHSP}}$ planner) domains for the other types of planning tasks.

Another conclusion we can draw from those results is

Table 1: Quality and scaling of satisficing planners YAHSP, LAMA, LPG, TFD and DAE$_{\text{YAHSP}}$ across the test domains. In column `Domain`(x), x denotes the total number of problem instances. Columns 2-4 (or 2-5) display the coverage, i.e., number of instances solved (and also, for DAE$_{\text{YAHSP}}$, the average number of successful runs — the closer to 11 the better). Columns 5-7 (or 6-9) show the quality score (and in parentheses, for DAE$_{\text{YAHSP}}$, the average coverage, the closer to the quality score the better). See text for the exact definitions. The values in bold are the best values obtained on each type of planning task (Cost, Temporal and STRIPS). Columns 8-10 (or 10-13) display the ratios $\frac{\text{Quality Score}}{\text{Coverage}}$ on each domain (with means of those ratios across the domain types).

Costs Domain-ipc6	Coverage			Quality			Quality / Total of solved problems		
	YAHSP	LAMA	DAE$_{\text{YAHSP}}$	YAHSP	LAMA	DAE$_{\text{YAHSP}}$	YAHSP	LAMA	DAE$_{\text{YAHSP}}$
Woodworking (30)	20	30	27 / 8.9	15.96	24.36	24.79 (24.3)	79.82%	81.21%	91.81%
Pegsolitaire (30)	30	30	30 / 10.9	20.90	26.19	28.11 (27.2)	69.66%	87.31%	93.71%
Parcprinter (30)	28	22	28 / 11	16.87	11.94	27.25 (17.0)	60.24%	54.27%	97.33%
Openstacks (30)	30	30	30 / 11	8.52	20.73	19.45 (18.2)	28.39%	69.12%	64.85%
Transport (30)	30	30	30 / 11	16.73	26.40	24.99 (23.0)	55.77%	88.00%	83.30%
Scanalyser (30)	27	30	27 / 11	13.68	25.88	21.85 (20.9)	50.66%	86.27%	80.92%
Elevator (30)	30	24	30 / 11	9.60	22.65	18.31 (16.3)	32.00%	94.36%	61.05%
Sokoban (30)	24	25	20 / 9.6	21.32	24.25	19.79 (19.3)	88.85%	97.02%	98.96%
Total problems (240)	*219*	*221*	***222***	*123.58*	*182.41*	***184.55***	*58.17%*	*82.19%*	***83.99%***

Temporal Domain	Coverage				Quality				Quality / Total of solved problems			
	YAHSP	LPG	TFD	DAE$_{\text{YAHSP}}$	YAHSP	LPG	TFD	DAE$_{\text{YAHSP}}$	YAHSP	LPG	TFD	DAE$_{\text{YAHSP}}$
Crewplanning-ipc6 (30)	30	12	29	30 / 11	24.55	12.00 (12.0)	28.76	29.90 (29.5)	81.82%	100%	99.17%	99.68%
Elevator-ipc6 (30)	30	30	17	30 / 11	8.31	25.83 (24.6)	13.45	23.24 (20.2)	27.70%	86.12%	79.11%	77.46%
Openstacks-ipc6 (30)	30	30	30	30 / 11	17.90	29.45 (27.6)	26.49	28.41 (27.8)	59.66%	98.15%	88.30%	94.71%
Pegsolitaire-ipc6 (30)	30	30	28	30 / 11	27.25	29.74 (28.4)	26.78	30.00 (29.8)	90.83%	99.14%	95.63%	100%
Parcprinter-ipc6 (30)	15	20	15	22 / 10.1	10.98	19.36 (19.2)	10.27	14.60 (14.2)	73.23%	96.82%	68.49%	66.35%
Sokoban-ipc6 (30)	22	16	17	17 / 10.5	17.20	11.14 (11.1)	12.74	15.60 (15.3)	78.20%	69.63%	74.92%	91.78%
Rovers-ipc3 (20)	20	20	6	20 / 11	17.74	19.95 (19.8)	5.78	19.86 (19.8)	88.69%	99.75%	96.39%	99.32%
Satellite-ipc3 (20)	20	20	20	20 / 11	6.33	20.00 (19.8)	12.55	16.86 (16.2)	31.64%	100%	62.77%	84.28%
Zeno-ipc3 (20)	20	20	20	20 / 11	9.70	18.98 (18.4)	11.62	17.50 (16.7)	48.49%	94.92%	58.09%	87.50%
Total problems (240)	*217*	*198*	*182*	***219***	*139.96*	*186.46*	*148.44*	***195.97***	*64.47%*	***93.84%***	*80.32%*	*89.01%*

STRIPS Domain	Coverage				Quality				Quality / Total of solved problems			
	YAHSP	LPG	LAMA	DAE$_{\text{YAHSP}}$	YAHSP	LPG	LAMA	DAE$_{\text{YAHSP}}$	YAHSP	LPG	LAMA	DAE$_{\text{YAHSP}}$
Airport-ipc4 (50)	20	46	37	44 / 9.8	19.47	4 2.37 (41.1)	35.58	40.34 (38.9)	97.35%	92.10%	96.16%	91.69%
Psr small-ipc4 (50)	50	9	50	50 / 11	47.65	9.00 (9.0)	50.00	49.96 (49.9)	95.30%	100%	100%	99.91%
Satellite-ipc4 (36)	28	36	32	27 / 11	16.42	35.98 (35.9)	30.25	26.57 (26.5)	58.66%	99.95%	94.54%	98.40%
Openstacks-ipc5 (30)	30	23	30	30 / 10.8	27.98	22.43 (22.3)	28.55	29.97 (29.8)	93.28%	97.52%	95.16%	99.89%
Rovers-ipc3 (20)	20	20	20	20 / 11	17.74	19.93 (19.9)	19.33	19.80 (19.7)	88.71%	99.65%	96.63%	99.02%
Zeno-ipc3 (20)	20	20	20	20 / 11	15.37	19.45 (19.6)	19.25	18.91 (18.5)	76.86%	97.27%	96.23%	94.54%
Freecell-ipc3 (20)	20	20	20	20 / 8.5	12.50	18.01 (17.9)	19.52	15.68 (14.0)	62.52%	90.05%	97.62%	78.39%
Pathways-ipc5 (30)	30	30	29	30 / 11	25.57	29.37 (29.0)	26.78	29.47 (20.4)	85.25%	97.91%	92.34%	98.25%
Total problems (256)	*218*	*204*	*238*	***241***	*182.72*	*196.56*	*229.26*	***230.70***	*82.24%*	***96.81%***	*96.09%*	*95.01%*

the robustness of DAE$_{\text{YAHSP}}$. Its coverage robustness is assessed by its very high average coverage (close to the maximum value 11): when an instance is solvable, almost all runs succeed. Regarding the quality robustness, the average quality of DAE$_{\text{YAHSP}}$ is most of the times greater than 95% of the quality score, with however some low value outliers. It is nevertheless difficult to compare, with respect to coverage, stochastic algorithms (like DAE$_{\text{YAHSP}}$ and LPG) to deterministic suboptimal ones (like LAMA). Lack of space forbids to present Cumulative Distribution Functions describing the distribution of the proportion of runs that did reach a given fitness value in a given time. Let us simply complement here the results of Table 1: replacing the coverage condition by requiring that the instance is solved at least 3 (resp. 6) times out of 11, DAE$_{\text{YAHSP}}$ is still slightly ahead of (resp. now slightly behind) LAMA. The general conclusion — DAEX performs as well as LAMA — nevertheless holds.

Related Work

Addressing the planning problem with an evolutionary algorithm, Genetic Planning, is not new but is usually done with a direct encoding of partial plans, i.e., individuals represent linear lists of actions, and is also usually restricted to classical planning like in (Westerberg and Levine 2001) or (Brié and Morignot 2005). A genetic algorithm for learning macro-actions for arbitrary planners and domains has been recently proposed in (Newton et al. 2007). In aggregating several steps, macros indirectly divide the state space by fostering better plan trajectories among all possible ones but the approach is much different from DAE$_{\text{X}}$. It is worth mentioning also a successful space application, modeled with timelines and a multi-objective function, reported in (Cesta et al. 2008) and in which the MRSPOCK solver includes a classical genetic algorithm. But although it is indeed a practical application of evolutionary computation to planning, the representation and operators used within MRSPOCK are very different from what is done in DAE$_X$.

LPG works by performing a stochastic local search, similar to WalkSat, on planning graph subsets (Gerevini, Saetti, and Serina 2003b). In both LPG and DAE$_X$, the strategy consists in gradually improving plan trajectories using a stochastic scheme. Other similarities are timestamping atoms with an earliest time estimate, and mutual exclusion constraints. However, there are fundamental differences between the two approaches. Firstly, LPG is a self-contained planner that performs a constructive method and reasons on partial plans, whereas DAE$_X$ is a meta-algorithm that modifies intermediate states and relies on an external solver to generate partial plans. Furthermore, although it manipulates several different plans by doing restarts, LPG is not a population-based search algorithm, because there is no interaction between the different "individuals". The use of timestamping is also very different in both approaches.

Plan optimization is also often performed by anytime search algorithms, such as LAMA; however, as mentioned in (Richter, Thayer, and Ruml 2009), such algorithms are often caught in unpromising parts of the search space, thus being unable to really improve the plan. They show that doing restarts in this kind of algorithms may be a better strategy. In contrast, our approach is designed to introduce diversity in the exploration of the search space, while taking benefit of the past exploration through the evolution of the population.

Conclusion

This paper introduced DAE$_X$, an evolutionary metaheuristic for satisficing planning. DAE$_X$ optimizes the decomposition of a planning task into a sequence of intermediate states that must be reached in turn by an embedded planner, in order to find a plan of the best possible quality. Creating the initial population and evolving the individuals from one population to the next through variation operators heavily relies on standard features of modern planners, such as binary mutual exclusions and reachability heuristics, in order to build time-coherent mutex-free partial states. Experiments demonstrate that the performance of an encapsulated planner can be greatly increased, both in terms of coverage and solution quality, making it competitive with state-of-the art planners. Although we used a single planner (YAHSP) in our experiments, future works will use different planners, evaluating their behavior within DAE$_X$. A portfolio of planners could also be used to solve each subtask; a sequence of solvers would then be recorded in the individuals. It is also interesting to see that these results are obtained with the simple h^1 planning heuristic for the construction of individuals; on the one hand, its use improved a lot the results; on the other hand, the use of more elaborate heuristics may be envisaged. A precise assessment of the impact of such heuristics on the results will be addressed in further work.

References

Bäckström, C. 1998. Computational aspects of reordering plans. *Journal of Artificial Intelligence Research* 9:99–137.

Bibai, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2008. DAE: Planning as Artificial Evolution (IPC-6 Deterministic part). `http://ipc.icaps-conference.org/`.

Bibai, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2009. Learning Divide-and-Evolve Parameter Configurations with Racing. In *ICAPS 2009 Workshop on Planning and Learning*.

Bibai, J.; Schoenauer, M.; and Savéant, P. 2009. Divide-And-Evolve Facing State-of-the-Art Temporal Planners during IPC-6. In EvoCOP'09, 133–144. Springer Verlag.

Brié, A. H., and Morignot, P. 2005. Genetic Planning Using Variable Length Chromosomes. In 15^{th} ICAPS, 320–329.

Cesta, A.; Cortellessa, G.; Fratini, S.; and Oddi, A. 2008. Looking for MrSPOCK: Issues in Deploying a Space Application. In *ICAPS 2008 SPARK Workshop*.

Chen, Y.; Hsu, C.; and Wah, B. 2006. Temporal Planning using Subgoal Partitioning and Resolution in SGPlan. *Artificial Intelligence* 26:323–369.

Eiben, A., and Smith, J. 2003. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer.

Eyerich, P.; Mattmüller, R.; and Röger, G. 2009. Using the Context-enhanced Additive Heuristic for Temporal and Numeric Planning. In 19^{th} ICAPS, 130–137.

Gerevini, A.; Saetti, A.; and Serina, I. 2003a. On Managing Temporal Information for Handling Durative Actions in LPG. In *AI*IA 2003: Advances in Artificial Intelligence*. Springer Verlag.

Gerevini, A.; Saetti, A.; and Serina, I. 2003b. Planning through Stochastic Local Search and Temporal Action Graphs in LPG. *JAIR* 20:239–290.

Haslum, P., and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In 5^{th} AIPS, 140–149.

Helmert, M. 2008. *Understanding Planning Tasks*. Springer Verlag.

Korf, R. 1987. Planning as Search: A Quantitative Approach. *Artificial Intelligence* 33:65–88.

Newton, M. H.; Levine, J.; Fox, M.; and Long, D. 2007. Learning Macro-Actions for Arbitrary Planners and Domains. In 17^{th} ICAPS, 256–263.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks Revisited. In AAAI'08, 975–982. AAAI Press.

Richter, S.; Thayer, J. T.; and Ruml, W. 2009. The Joy of Forgetting: Faster Anytime Search via Restarting. In SOCS'09.

Schoenauer, M.; Savéant, P.; and Vidal, V. 2006. Divide-and-Evolve: a New Memetic Scheme for Domain-Independent Temporal Planning. In EvoCOP'06, 247–260.

Schoenauer, M.; Savéant, P.; and Vidal, V. 2007. Divide-and-Evolve: a Sequential Hybridization Strategy using Evolutionary Algorithms. In Michalewicz, Z., and Siarry, P., eds., *Advances in Metaheuristics for Hard Optimization*, 179–198. Springer Verlag.

Sebastia, L.; Onaindia, E.; and Marza, E. 2006. Decomposition of Planning Problems. *AI Communications* 19(1):49–81.

Vidal, V., and Geffner, H. 2006. Branching and Pruning: An Optimal Temporal POCL Planner based on Constraint Programming. *Artificial Intelligence* 170(3):298–335.

Vidal, V. 2004. A Lookahead Strategy for Heuristic Search Planning. In 14^{th} ICAPS, 150–160.

Westerberg, H., and Levine, J. 2001. Optimising Plans using Genetic Programming. In 6^{th} *Eur. Conf. on Planning (ECP-01)*.

Yuan, B., and Gallagher, M. 2004. Statistical Racing Techniques for Improved Empirical Evaluation of Evolutionary Algorithms. In PPSN VIII, 172–181.