Evolutionary State Decomposition for Temporal Planning

Pierre Savéant\textsuperscript{1} Marc Schoenauer\textsuperscript{2} Jacques Bibaï\textsuperscript{1,2} Vincent Vidal\textsuperscript{3}
\textsuperscript{1}Thales Research & Technology, Palaiseau, France  \textsuperscript{2}Projet TAO, INRIA Saclay & LRI, Université Paris Sud, Orsay, France  \textsuperscript{3}CRIL & Université d’Artois, Lens, France
first.last@thalesgroup.com first.last@inria.fr vidal@cril.univ-artois.fr

Abstract

Applying a Divide-and-Conquer strategy for solving Planning Problems has been investigated in various deterministic approaches in order to reduce time complexity, though sometimes at the cost of a lower quality solution plan. This paper presents a stochastic method for Planning Decomposition which focuses on plan quality. The scope is Temporal Planning as defined by PDDL2.1, where the problems are described using actions with duration and where plan quality is the total makespan. An evolutionary algorithm learns tractable sequences of sub-problems without using any structural properties of the problem, and a compression routine further optimizes the simple concatenation of the sub-plans. The approach is demonstrated on IPC-6 domains, and compared to both SGPLAN, the winner of the competition, and to the BASELINE planner provided by the organizers. Results validate the concept and show quality improvement.

1 Introduction

Divide-and-Conquer is a generic strategy that has been applied to Planning Problems in different works, that usually deterministically exploit properties of the domain or problem structure in order to identify independent subproblems. In [12] for instance, the division is based on ordered landmarks [5] whereas in SGPLAN [2], subgoal partitioning is based on mutual-exclusion constraints which show strong locality in most of IPC benchmarks. The objective of these approaches is to reduce time complexity, though sometimes at the cost of a lower quality solution plan.

As stated in [12], a Planning Decomposition method requires (a) a decomposition principle, (b) an algorithm to solve the subproblems individually and (c) a procedure to recombine the subplans into a global solution plan.

This paper builds on a stochastic approach for Planning Decomposition termed Divide-and-Evolve (D&E) [10; 11] by adding domain knowledge at different steps of the algorithm. The basic idea of D&E is to search the space of state decompositions of the planning problem at hand by means of Artificial Evolution: candidate solutions are sequences of intermediate goals which define consecutive planning subproblems that are hopefully easier to solve than the global problem. The (b) issue, solving the individual subproblems, is addressed by handling the subproblems over to a black-box subplanner. Though any planner could a priori be used, the choice of CPT [14] was motivated not only for its efficiency as an optimal temporal planner, but also because of its internal representation, that helps solving the (c) issue, subplan recombination. Indeed, thanks to the constraint-based representation of CPT and its Partial Order Causal Link (POCL) planning strategy [7], an efficient compression routine that exploits causal links and precedence constraints has been implemented. In comparison, most of temporal planners (e.g. SGPLAN) use a two-stage approach by first ignoring durations and concurrency, and then rescheduling afterwards, with the risk of missing some globally optimal plans.

The other originality of D&E is that it uses Evolutionary Algorithms (EAs), a ‘blind’ stochastic approach to overcome the limitations of methods based on the First Principles (i.e. theoretically wee-founded) to address issue (a) and perform the actual decomposition. There are indeed many examples of situations where stochastic search can perform better than exact methods, as witnessed for instance every year at the Humies Awards of the GECCO conferences. However, though ‘blind’ with respect to the objective, all such successful approaches incorporate some specific knowledge about the problem at hand. But this knowledge is used to improve the efficiency (e.g. decreasing the size of the search space), but not to directly guide the search. In some sense, Evolutionary methods use knowledge in a declarative way, as opposed to the procedural way that would bias and hence possibly hinder the search.

This paper introduces DAE2, that adds specific domain knowledge to the first versions of D&E. First, only intermediate goals containing pairwise consistent atoms are explored. Furthermore, the initialization of the population (i.e. the random generation of state decompositions) is made according to lower bounds on the minimum time needed to make a given atom true from the initial situation [4]. Another important innovation is a new multi-step approach to the compression stage.

DAE2 has participated to the IPC-6 ’deterministic track’ competition: whereas hindered by the tight time limit, the quality of the solution plans it obtained on all instances it could solve was better than that of its competitors. Further
experimental results on the IPC-6 domains are presented here and compared, with respect to plan quality, first to those of CPT alone, then to those of both SGPLAN, the winner of the competition [2] and the very good BASELINE planner kindly provided by Malte Helmert, one of the organizers.

Next Section describes the AI Planning background, and introduces the original compression method built on top of CPT. Section 3 details the different components of DAE2, while Section 4 presents and discusses the experimental results of DAE2 and Section 5 concludes the paper.

2 Rationale

The scope of this work is temporal planning as defined by PDDL2.1 [3] restricted to simple grounded temporal problems, following the semantics of temporal planning defined in [13], where conflicting actions cannot overlap over time.

A Temporal Planning Problem is a tuple \( P = (A, O, I, G) \), where \( A \) is a set of atoms representing all the possible facts in a world situation, \( O \) is a set of actions, and \( I \) and \( G \) are two sets of atoms that respectively denote the initial state and the temporal planning goal. An action is a tuple \( a = (\text{pre}(a), \text{add}(a), \text{del}(a), \text{dur}(a)) \) where \( \text{pre}(a), \text{add}(a) \) and \( \text{del}(a) \) are sets of atoms that respectively denote the preconditions, adds effects and del effects of \( a \), and \( \text{dur}(a) \) denotes the duration of \( a \), a rational number.

A feasible plan \( P \) to the planning problem \( \langle A, O, I, G \rangle \) is a list of actions such that all actions can be applied in turn (eventually in parallel) starting from the initial state \( I \), and result in setting all atoms of the goal to true. The quality of a plan is usually the total duration of its sequence of actions, or makespan, and a solution to \( P \) is a plan with minimal makespan.

This work will also use the notion of mutex pair of atoms, as defined in the context of CPT [14]: \( \{a, b\} \) is mutex if \( a \) and \( b \) cannot be true at the same time – \( M(a) \) will denote the set of all \( b \in A \) s.t. \( \{a, b\} \) is a mutex.

2.1 State Decomposition

The aim of state decomposition is to transform a given planning problem into a sequence of hopefully easier subproblems. Indeed, temporal planning introduces concurrency among actions, involving in turn resource sharing issues. Among the multiple alternative plan trajectories, combinatorial explosion in temporal planning results for instance from task ordering permutations, that produce equivalent quality plans but are nevertheless searched for by most planners. At the opposite, when multiple resources can be used concurrently (or multiple actions can be driven concurrently), an excessive decomposition might miss potential parallelism, that hence needs to be recovered by the compression stage (c).

A general decomposition of a planning problem \( P = \langle A, O, I, G \rangle \) is hence simply a sequence of intermediate states \( S = (s_i)_{i \in [0..n+1]} \), of unknown length \( n \), as no a priori assumption is made on the number of states, and where \( s_0 = I \) and \( s_{n+1} = G \). Once given such a sequence \( S \), solving \( P \) amounts to finding in turn a solution \( P_i \) for all subproblems \( \sigma_i = \langle A, O, s_i, s_{i+1} \rangle (i = 0, \ldots, n) \) (step (b) in [12]).

Indeed, after solving all \( P_i \), the concatenation of all subplans \( P_i \) is already a solution to the global problem. However, this solution is likely to be of poor quality since concurrency among actions might be missed. A compression step of such a naive concatenation is hence needed (the step (c) in [12]). Unfortunately, compressing a set of subplans into a global plan can be a difficult problem per se with respect to the expected level of compression.

2.2 Subplan compression

The main reason for choosing CPT is probably its internal representation of partial plans, which made it possible to design a fast and efficient compression routine based on causal links and precedence constraints.

As usual in POCL planning [7], a partial plan is represented in CPT by a tuple \( \langle \text{Steps}, \text{Ord}, \text{CL}, \text{Open} \rangle \) where \( \text{Steps} \) is the set of actions in the partial plan, \( \text{Ord} \) is a set of precedence constraints on \( \text{Steps} \) of the form \( a_1 < a_2, \text{CL} \) is a set of causal links of the form \( a_1(p)\lnot a_2 \) where \( a_1, a_2 \in \text{Steps} \) and \( p \in \text{add}(a_1) \cap \text{pre}(a_2) \), and \( \text{Open} \) is a set of open conditions (preconditions or goals not supported yet by a causal link of \( \text{CL} \)). \( \text{Steps} \) also contains two dummy actions \( \text{Start} \) and \( \text{End} \) with zero durations, the first with an empty precondition and effect the initial state of the problem; the latter with precondition the goal of the problem and empty effects. These actions are used to initialize the first partial plan given to the planner, that will be refined up to a solution. A solution plan is a partial plan with no open condition and no flaws, which can be either causal links threatened by actions in the plan that must be protected by precedence relations, or unordered effect-conflicting actions.

Let \( P = \langle A, O, I, G \rangle \) be a temporal planning problem and \( (s_i)_{i \in [0..n+1]} \) be a decomposition of \( P \), given by the evolutionary algorithm (EA), that admits a solution. Then \( s_0 = I, s_{n+1} = G \) and the dummy actions \( \text{Start} \) and \( \text{End} \) are such that \( \text{add(Start)} = I \) and \( \text{pre(End)} = G \). To each partial state \( s_i \) with \( i \in [1..n+1] \), is associated a partial plan (computed by CPT) \( \sigma_i = \langle \text{Steps_i}, \text{Ord_i}, \text{CL_i}, \text{Open_i} \rangle \) with dummy actions \( \text{Start_i} \) and \( \text{End_i} \) such that \( \text{add(Start_i)} = \text{End_i} \). As the decomposition admits a solution, all these partial plans are solution plans of their respective subproblems and are such that \( \text{Open_i} = \emptyset \). The compression routine simply consists in feeding CPT with an initial partial plan \( \sigma = \langle \text{Steps}, \text{Ord}, \text{CL}, \text{Open} \rangle \) such that:

\[
\text{Steps} = \bigcup_{i=1}^{n+1} \{\text{Steps}_i \setminus \{\text{Start}_i, \text{End}_i\} \} \cup \{\text{Start}, \text{End}\}; \\
\text{Ord} = \bigcup_{i=1}^{n+1} \{a_1 \prec a_2 \in \text{Ord}_i \mid a_1 \neq \text{Start}_i \land a_2 \neq \text{End}_i\}; \\
\text{CL} = \bigcup_{i=1}^{n+1} \{a_1(p)\lnot a_2 \in \text{CL}_i \mid a_1 \neq \text{Start}_i \land a_2 \neq \text{End}_i\}; \\
\text{Open} = \bigcup_{i=1}^{n+1} \{p \mid \text{Start}_i(p) \in \text{CL}_i \} \cup G.
\]

CPT then has to solve \( P' = \langle \{f \in A \mid \exists a \in \text{Steps}, f \in \text{pre}(a) \cup \text{add}(a) \cup \text{del}(a)\}, \text{Steps}, I, G \rangle \), starting from the initial partial plan \( \sigma \). Furthermore, CPT is run in its ‘canonical’ mode, making the compression problem close to a scheduling problem: all the actions of \( \text{Steps} \) already belong to the partial plan, no new action can be used, so only a valid schedule respecting at least the constraints of \( \text{Ord} \) and \( \text{CL} \) has to be found. Optimally solving this problem is NP-complete: in the worst case, all subplans contain one action, no precedence relation or causal link can be used, and all possible schedules
between actions have to be considered. But this problem is not \text{PSPACE}-complete: the makespan of the global plan is polynomially bounded by the sum of the subplan makespans, as their simple concatenation is a suboptimal solution of $P'$.

In summary, the compression step can be seen as a new solving process, more or less informed by deductions collected during the solving of each subproblem.

3 Divide-and-Evolve-2

This section presents the details of the implementation of an Evolutionary Algorithm to solve the optimization problem introduced in previous Section, as well as the embedded planner used here, CPT.

As advocated in [12], the first ingredient for state decomposition is a decomposition principle. Previous works have tackled this issue by relying on First Principles. On the opposite, D&E addresses the problem of finding a decomposition of $P = \langle A, O, I, G \rangle$ by turning it into an optimization problem: search for a sequence $S = \langle s_i \rangle_{i \in [0, n+1]}$ such that the plan $\sigma$ obtained by compressing subplans $\sigma_i$ found by CPT as solutions of $P_i = \langle A, O, s_i, s_{i+1} \rangle_{i \in [0, n]}$ has minimal makespan. 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 proved 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 D&E: avoid unnecessary restrictions on the search space or the objective function, and use very flexible, though powerful, optimization algorithms: Evolutionary Algorithms.

Evolutionary Algorithms (EAs) are general purpose optimization algorithms that have been demonstrated to be flexible and robust enough to handle such challenging optimization problems. In particular, several EA successes have been obtained in similar contexts of unstructured spaces, e.g. parse trees in the case of Genetic Programming for Analog Circuit Design [6], or Voronoi Diagrams for Structural Design Applications [8].

3.1 Representation

Individuals must here represent state decompositions, i.e., in order to stay as general as possible, variable length lists of states. However, because the space of all possible states is huge, and because an intermediate state of a decomposition is to be used as a goal for a subproblem, it is possible to describe it only partially: limiting the set of atoms that will be used to describe an intermediate state will greatly reduce the size of the search space. But this raises the additional issue of the choice of the atoms to be used to represent individuals? Whereas this was left to the user in the original D&E, the intermediate states of DAE2 will use all atoms that are based on the same predicate that at least one atom of the goal $G$.

Another crucial issue is that of state consistency w.r.t. the set of actions. Ideally, we should avoid the generation of unreachable states. However, since a complete check of state consistency has the same complexity than solving the corresponding subproblem, some trade-off has to be considered. In the original D&E, the user again was asked to manually designate inconsistent pairs (e.g. the same person cannot be in 2 different cities at the same time). DAE2 will only consider intermediate states in which atoms are pairwise non-mutex, with the help of CPT data structures (see next Section 3.2).

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. Unfortunately, the only way to ensure that would be to actually run the planner, as we are not aware of any metric between states such that difficulty of a planning problem would be correlated with the distance between the initial state and the goal. The workaround used for D&E is to use a purely syntactic (asymmetric) metric $d_c$: for any complete state $s$ and partial state $g$, $d_s(s, g)$ is the number of atoms in $g$ that are not true in $s$.

3.2 Fitness Computation

Two different situations should be distinguished here, depending on whether or not the sub-planner fails on one of the subproblems or during the compression step, in which case the decomposition is termed unfeasible. First, all unfeasible individuals should get a worse fitness than all feasible ones; Second, there should be some fitness pressure toward feasibility for unfeasible individuals.

Pseudo-code for the computation of the fitness is given in Algorithm 1. The main loop (lines 4-14) processes the intermediate states sequentially, calling CPT to solve the corresponding planning sub-problem (line 7): The initial state is the current state $cs$, computed by actually running the solution plan of the previous subproblem (line 13); The goal is the currently processed intermediate state $s$; The last argument $b_{max}$ is a limit on the maximum number of backtracks that CPT is allowed per iteration. Indeed, because there can be no guarantee that the sub-problems are easier to solve than the original one, it is mandatory to restrain CPT from a too deep (and long!) exploration. This limit $b_{max}$ is computed (line 6) according to the following empirical formula, derived from numerous experiments on IPC-3 benchmarks:

$$b_{max} = #G \ast \left( \frac{\#nodes}{\#conflicts} + 2 \ast \frac{\#causals + \#actions}{\#atoms} \right) \quad (1)$$

where $#G$ is the number of goal atoms, $\#causals$ the number of causal links, $\#actions$ the number of actions, and $\#atoms$ the number of atoms generated after action grounding. Those numbers are easily computed from the domain and instance descriptions. Further experiments have also demonstrated the need for more backtracks when solving the last subproblem (reaching the problem goal): in this case a multiplicative factor of 7 is added to Equation (1). Finally, the number of backtracks is in any case limited to 1000000.

CPT returns $sol_k$, the (optimal) solution of the current subproblem, together with its makespan $m$, and the number of backtracks $b$ that was needed to find it – unless it fails within the maximum number of backtracks $b_{max}$ and returns $FAIL$.

In the latter case, the fitness is set according to line 9: it aims at minimizing the syntactic distance $d_c(cs, G)$ between the Goal and the current initial state $cs$, that is also the last current complete state that has been reached: after a subproblem has been solved, this current complete state is obtained
by running the optimal plan (line 13). 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 strictly positive makespan to be reached (line 10). \( \alpha \) is a large enough constant that ensures that all feasible individuals get a larger fitness than all feasible ones.

When the individual is feasible (all subproblems are solved by CPT), the compression routine is used to compress all sub-plans (line 15), and the fitness is basically the total makespan \( M \) of the resulting global plan. However, as in the unfeasible case, it was necessary to penalize the individual by the amount of useless intermediate states, in order to avoid unnecessary bolt. Furthermore, a second additional term favors “easy” sub-problems by penalizing all problems with the cumulated number of backtracks \( B \) actually used by CPT, leading to the formula of line 17.

One improvement of DAE2 is its sequential use of different types of information during the compression (Section 2.2): the more information (i.e., constraints) it is given about the sub-plans, the simpler the compression problem, but the lower the quality of the compressed plan. Here, process starts with only the actions. If it fails (within the limit of allowed backtracks), the precedence constraints are added. If it still fails, the causal links are added: at this point, it has always returned a good compressed plan using almost no backtrack.

3.3 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 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.

The crossover operator is the basic 1-point crossover for variable length representations: in order to recombine \((s_i)_1 \leq n\) and \((t_i)_1 \leq m\), it chooses uniformly some states \(s_a\) and \(t_b\), and exchanges the second parts of both lists, obtaining the two offspring \((s_1, \ldots, s_a, t_b+1, \ldots, t_m)\) and \((t_1, \ldots, t_b, s_a+1, \ldots, s_n)\).

Four different mutation operators have been used. Assume parent is \((s_1, \ldots, s_{last}, \ldots, s_n)\), where \(s_{last}\) is the last state reached by the sub-planner. At the individual level, mutation addStation randomly adds a state after state \( j \leq min(n, last) \) as described in Algorithm 2: this new intermediate state will contain all common atoms of \(s_j\) and \(s_j+1\), half of the other atoms of either \(s_j\) or \(s_{j+1}\) (but ensuring the pairwise non-mutex compatibility), and one randomly chosen atom. Reciprocally, mutation delState removes state \(s_i\), with \(i\) uniformly chosen in \([1, min(n, last + 1)]\). At the station level, mutation changeAtom modifies one random atom in each state \(s_i\) \((i \in [1, min(n, last + 1)])\), modifying randomly one of its arguments while maintaining the pairwise consistency, and mutation delAtom removes one uniformly chosen atom from state \(s_i\), with \(i\) uniformly chosen in \([1, min(n, last + 1)]\).

3.4 Initialization of the population

The initialization in DAE2 is heavily based on the lower bounds on the earliest time an atom \(a\) can become true, denoted \( h^2(a) \), and computed using a relaxation of the initial problem as proposed in [4]. The pseudo-code for the initialization is given in Algorithm 3: it chooses some dates among the possible lower bounds for atoms of the goal, and for each date, builds an intermediate state with one random set of atoms that can have become true earlier, ensuring each state contains no mutex pairs.

4 Experimental results

Divide-and-Evolve has been compared first to CPT alone, and also to two of the best temporal planners that entered the 6th International Planning Competition (IPC-6): SGPPLAN6 was the overall winner, and BASELINE was proposed by the organizers, thus not taking part to the competition, but sometimes reporting better results than SGPPLAN6. Both codes were downloaded from the competition site.

Six domains were proposed during the competition. However, two of them were too large for CPT to even start running its initialization steps. Hence the comparison has
been performed on the 4 domains openstacks, crew planning, parc printer, and peg solitaire.

Whereas the running time of all algorithms was limited to 30mn per instance during the competition, all algorithms were given here an unlimited amount of time, even if SG-ASELINE requires respectively 12, 5 and 3 millions of backtracks, while D&E requires respectively 12, 5 and 3 millions of backtracks, while D&E requires respectively 12, 5 and 3 millions of backtracks.

Furthermore, the stochastic nature of DAE2 requires several independent runs to be made for any conclusion to be drawn: all experiments presented here are based on 11 such runs, and all statistical analyzes, as well as all following plots, have been made using the Open Source R package.

4.1 Divide-and-Evolve Settings

One identified weaknesses of EAs is the difficulty in tuning their numerous parameters, as there exist 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).

For DAE2, some parameters were set once and for all, based on previous findings [11]. This is the case for all components of the Artificial Darwinism part, set to a (10+70)-ES: 10 parents generate 7 offspring each using the variation operators, and the best of those 80 individuals become the parents of the next generation. The same stopping criterion has also been used for all experiments: evolution stops when no improvement in fitness occurs in 10 generations, with a maximum of 100 generations altogether.

The parameters regarding the application of the variation operators are the crossover and mutation probabilities \( p_c \) and \( p_m \), and the choice of which mutation to apply is made by a roulette-wheel tournament governed by 4 weights \( w_1, \ldots, w_4 \) for each of the 4 mutations (see Section 3.3). A two-stage DOE was used: first, the \( w_i \) were set to \( (4, 1, 4, 1) \), and an incomplete factorial DOE was done on \( (p_c, p_m) \) using IPC-3 benchmarks. The differences were validated using both Kolmogorov and Wilcoxon non-parametric tests at 95% confidence levels. Three \( (p_c, p_m) \) pairs were found significantly better, and a complete factorial DOE on the 4 weights and those 3 pairs yielded the final setting: \( (0.4, 0.9) \) for \( (p_c, p_m) \), and \( (35, 3, 35, 7) \) for the \( (w_i) \).

4.2 Results

Figures 1 to 4 display, for all 3 algorithms, the makespan for all instances of the 4 domains. Each column corresponds to an instance (number on the X axis). For the deterministic SGPPLAN6 and BASELINE, symbols (‘@’ and ‘#’ respectively) indicate the best makespan found before stopping. For the stochastic DAE2, standard boxplots sketch the distribution of the 11 makespans. For CPT, a small “*” at the bottom of a column indicates that CPT could solve this instance on its own (obtaining the same fitness that the best of DAE2 runs).

The first conclusion is that DAE2 solves almost all instances (exceptions are crew planning 23, 29 and 30 and parc printer 18), whereas CPT fails on many of them (e.g. on openstacks 8-30). Note that the original version of D&E is not mentioned here, as it performed worse than DAE2. Furthermore, comparing the number of backtracks needed by CPT and the cumulated number of backtracks from all CPT calls within the computation of the fitness of the best solutions found by DAE2 (\( B \) on line 12 in Algorithm 1), there is indeed a huge improvement in complexity. For instance, on openstacks 5, 6, and 7, CPT requires respectively 12, 5 and 3 millions of backtracks, while the avg. \( B \) values for DAE2 are 210, 3900 and 430.

Moreover, DAE2 always provides better quality results than the competing algorithms, and in addition provides them reliably (dispersions of the makespans are small). Of course, DAE2 did use much more time than 30mn, as illustrated on Figure 5, showing the distribution of run-times per instance of openstack), going up to a few hours for the most difficult

---

### Algorithm 3 Initialization

```plaintext
Require: \( T = \{h^2(a) : a \in G\} \) // See [4]
Require: \( \{M(a) : a \in G\} \) // All mutex of \( a \\
1: N \leftarrow \text{IU}[1, \#T] \) // number of intermediate states
2: \( D \leftarrow \{\} \) // list of dates
3: for \( i = 0 \) to \( N \) do
4: \( t \leftarrow \text{IU}(T) ; T \leftarrow T \setminus \{t\} \)
5: Insert(t, D) // D is ordered
6: Individual \leftarrow \{\} // Start building the individual
7: for \( t \) in \( D \) do
8: \( s \leftarrow \{\} \) // Start building the intermediate state
9: \( S \leftarrow \{a \in G : h^2(a) \leq t\} \) // atoms that can be true at \( t \)
10: while \( (S \neq \{\}) \) do
11: \( a \leftarrow \text{IU}(S) \) // Uniform choice in \( G \)
12: \( s \leftarrow s + \{a\} \) // Add to \( s \)
13: \( S \leftarrow S \setminus (\{a\} \cup M(a)) \) // remove all mutex(a)
14: Individual \leftarrow Individual + \{s\}
15: return Individual
```

---

**Figure 1:** Comparative results on openstack domain for DAE (boxplots), SGPPLAN6 (@), and BASELINE (#). See text for details.
Figure 2: Comparative results on crew planning domain for DAE (boxplots), SGPLAN6 (@), and BASELINE (#). See text for details.

Figure 3: Comparative results on parc printer domain for DAE (boxplots), SGPLAN6 (@), and BASELINE (#). See text for details.

Figure 4: Comparative results on peg solitaire domain for DAE (boxplots), SGPLAN6 (@), and BASELINE (#). See text for details.

Figure 5: Boxplots of the computational cost (seconds) of DAE2 for the 11 runs per instance of Figure 1 (on openstacks domain). For the sake of comparison, though not represented here for the sake of readability, the running times of the BASELINE Planner are lower than 1 second and those of SGPLAN6 range from a few seconds to 25000 seconds.
Figure 6: Effects of compression on openstacks domain: for each instance, the upper boxplot represents the makespans (over 11 runs) of the best solution before compression, and the lower boxplots the makespans of the same solutions after compression. The lower boxplots are identical to the DAE2 boxplots of Figure 1.

instances. This is certainly unfair to the other planners, that might have been tuned to use more time, and maybe obtain better quality results. However, DAE2 also obtains the best results for all the small instances that it solves in less than 30mn.

The effectiveness of the compression step is assessed on Figure 4.1 that shows, for each instance of the openstack domain, boxplots of the distribution of the makespans of the best plan before and after compression. This demonstrates that the compression routine is effective (other domains yield similar plots), and that its effectiveness increases with the difficulty of the instance. Furthermore, one can also see that the makespans before compression have a much larger dispersion than after compression: indeed, many different decompositions are equivalent after compression. But this in turn could hinder the search, creating large plateaus of identical makespans, a posteriori justifying the additional terms in the fitness (Algorithm 1 line 9).

Figures ?? and 8 displays boxplots for respectively the number of stations and atoms per station for the best decompositions obtained by DAE2 on openstacks domain (the situation is similar on other domains). It shows that DAE2 builds larger decompositions with more atoms per state as instances get harder – even though the settings are the same for all instances. DAE2 thus seems to somehow grasp instance difficulty.

Figure 7: Diversity of DAE2 results on openstacks domain: Boxplots of number of intermediate states in the best solutions for each instance.

5 Discussion and Conclusion
Decomposition is a way to overcome the curse of complexity of Temporal Planning Problems. This paper has introduced a refined version of DAE2, a method that successfully calls upon both stochastic search to explore the space of state-space decomposition and sound principles of constrained programming to first solve the subproblems and then recombine and compress the partial plans thus obtained.

To the best of our knowledge, and rather surprisingly, there have been very few attempts in the past to apply evolutionary algorithms to planning problems, and most works use a direct encoding of partial plans, such as [1], that obviously does not scale up well. A genetic algorithm for learning macro-actions for arbitrary planners and domains has been recently proposed [9]. Combining several steps, macros indirectly divide the state space by favoring better plan trajectories among all possible ones. But the goal there is to learn macro-actions on small instances and use then on larger ones, not to come up with a generic approach that focuses on plan quality.

We believe that the key to DAE2 performance and reliability is the use of domain knowledge to (a) restrict the space of possible states by ensuring pairwise consistency of atoms, from initialization to variation operators; (b) feed the initial population with states that at least are possible according to the lower bounds derived in [4]; (c) optimize the recombination of subplans in reasonable time, thanks to the data structures maintained by CPT. The drawback of the good quality of the solution plans in terms of total makespan is the high, but nevertheless manageable, CPU cost. However, in a design context where it is important to obtain a very good solution at least once, cost matters less, and DAE2 is a reliable alterna-
Figure 8: Diversity of DAE2 results on openstacks domain: Boxplots of number of atoms in the best solutions for each instance.

References


