FLAP: Applying Least-Commitment in Forward-Chaining Planning

Oscar Sapena, Eva Onaindia and
Alejandro Torreño
Depto. Sistemas Informáticos y Computación
Universitat Politècnica de València, Spain
E-mail: osapena@dsic.upv.es

In this paper, we present FLAP, a partial-order planner that accurately applies the least-commitment principle that governs traditional partial-order planning. FLAP fully exploits the partial ordering among actions of a plan and hence it solves more problems than other similar approaches. The search engine of FLAP uses a combination of different state-based heuristics and applies a parallel search technique to diversify the search in different directions when a plateau is found. In the experimental evaluation, we compare FLAP with OPTIC, LPG-td and TFD, three state-of-the-art non-linear planners. The results show that FLAP outperforms these planners in terms of number of problems solved; in addition, the plans of FLAP represent a good trade-off between quality and computational time.

Keywords: planning, partial-order plan, least commitment, forward-chaining search, heuristics

Introduction

Until the late 90s, Partial-Order Planning (POP) was the most popular approach to AI planning. POP follows the least-commitment principle by which decisions about action orderings and parameter bindings are postponed until a decision must be taken. This is an attractive idea as avoiding premature commitments requires less backtracking during the search. Nevertheless, the most recent state-based forward planners, such as LAMA [28] or SGPlan [6], have shown to be more efficient than partial-order planners, because (1) state-based planners can benefit from the existence of powerful heuristics and (2) generating a plan is far more costly than generating a state due to the need of conflict-checking mechanisms.

However, the general move towards state-based forward search ignores some important benefits of partial-order planning:

- A partial-order plan offers more flexibility in execution.
- The search can be easily guided to reduce the plan length; i.e., exploit the parallel execution of actions.
- It is a very suitable approach in multi-agent planning systems, either with loosely [21] or tightly coupled [32] agents.
- It can be easily extended to deal with temporal planning [1].

These desirable properties have led many researchers to adopt a POP approach and have motivated the revival of the investigation on partial-order planning.

The objective of this paper is to present FLAP, a partial-order forward chaining planner that follows the least-commitment strategy of POP except the delayed parameter binding. Unlike other planners, FLAP fully exploits delaying commitment to the order in which actions are applicable, thus achieving flexibility, reducing the need of backtracking and minimizing the length of the plans by promoting the parallel execution of actions. Although all these advantages come at an increase of the computational cost, FLAP applies an effective parallel search technique that allows solving more problems than other partial-order planners and returns plans that represent a good trade-off between quality and time in many domains.

In the remainder of the paper we present some related work and background, the planning approach of FLAP and a discussion on its limitations and possible extensions. Finally, we present an empirical evaluation of FLAP versus other partial-order planners, and we conclude with some final remarks.

Related work

With the aim of preserving POP benefits without sacrificing performance, some recent works focus on the generation of partial-order plans in forward-planning frameworks. The new planners that arise within this mixed framework have relaxed (or aban-

AI Communications
ISSN 0921-7126, IOS Press.

doned) the least-commitment strategy. Particularly, most POP planners work with fully instantiated actions rather than delaying parameter binding. This has been a predominant trend since the great success of Graph-Plan [2], a graph-based planning approach which computes all ground (fully instantiated) atoms and actions in a pre-processing stage.

Practicing a least-commitment strategy also implies recording only the essential action orderings, but this aspect has also been relaxed to a certain extent. One of the first works in this direction was FLECS [34]. This planner combines delayed and eager operator-ordering commitments. It can vary its commitment strategy across different problems and also during the course of a single planning problem. Since sequential execution is assumed, committing to an arbitrary total orderings is not considered harmful if it improves planning performance.

The MAPL [4] framework for temporal multi-agent planning subsumes a partial-order plan structure within a forward-search algorithm. MAPL records state information in a partial plan like the achieved state variable assignments. This information is called the *frontier state*, as it is built on the frontier of a partial-order plan. Additionally, it keeps the achiever of each variable assignment in order to check the plan validity. MAPL only considers the actions whose preconditions hold in the frontier state, so it can efficiently determine the actions that can be inserted in the current plan. Each new action is inserted after its precondition achievers and additional constraints are added to ensure that potential threats are resolved. The resulting algorithm is sound but incomplete.

More recently, the POPF planner [7] was developed following a similar approach to MAPL. POPF uses a partial-order plan construction within a forward-planning framework, working with time, numbers and continuous effects. Particularly, it inherits the forward-chaining search from CRIKEY [8], which works similarly to FF [17]. POPF records state information in each step of the plan (frontier state), like the negative interactions among the variable assignments, and updates the state accordingly. Like MAPL, the frontier state is used to determine the set of applicable actions at each step of the plan. The late-commitment approach of POPF is based on delaying commitment to ordering decisions on the frontier state, thus ignoring other alternative choices that would come earlier, i.e. *before* the frontier state. Completeness, however, is ensured as search performs backtracking to find an alternative plan when necessary.

OPTIC, the latest version of POPF, also handles soft constraints and preferences [1]. OPTIC has demonstrated to be one of the most relevant state-of-the-art planners in many domains. The key of its good performance is the fast generation of the successor states during the search and the use of effective domain-independent heuristics. OPTIC follows the same partial-order forward-planning approach of POPF. The frontier state information is used to add temporal constraints over the action only if they are required to ensure that preconditions are met. This approach represents a compromise between the total-ordering commitment of standard forward search and the least-commitment approach in partial-order planning, since it only commits to ordering choices that ensure consistency of the plan.

Combining a forward-search and a partial-order construction is a flexible approach that reports a very good performance, as OPTIC has demonstrated. This success is achieved by means of an eager parameter binding and an eager commitment to some ordering decisions to reduce the search overhead. Despite the success of this mixed approach, in this paper we want to address the following question: is it necessary to give up the application of the least-commitment strategy to guarantee a successful POP performance? Delaying decisions about when individual actions are to be scheduled is convenient in several contexts. The MAP-POP [32] multi-agent planner, for example, needs the ability to add new actions at any step of the current plan, not just when their preconditions hold in the frontier state, because this is the mechanism that agents use to tell the others they can contribute in the construction of the joint plan. On the other hand, delaying commitment of choices on the ordering of the actions reduces the need of backtracking.

Background

In this section, we provide some definitions of planning concepts, partial-order planning and landmarks, which will be used throughout the manuscript.

Preliminary notions on planning

In this work, we use a state-variable representation of the planning problem instead of the traditional propositional representation. Specifically, the planning language used in FLAP is based on PDDL3.1 [20], the latest version of PDDL. Unlike the previous PDDL

versions, which model planning tasks through predicates, PDDL3.1 incorporates state variables that map to a finite domain of objects of the planning task.

A planning task is a tuple $T = \langle V, A, I, G \rangle$. V is a finite set of state variables, each of which is associated to a finite domain, D_v, of mutually exclusive values that refer to the objects of the planning task. A is the set of deterministic actions of the agent. I is the set of initial values assigned to the state variables in V, and represents the initial state of the task T. G is the set of goals of the task, i.e., the values that the state variables are expected to take in the final state.

Variables in V are used to model the states of the world (problem states). When a value is assigned to a state variable, the pair $\langle$variable, value$\rangle$ acts as a ground atom in propositional planning.

Definition 1. (Fluent) A ground atom or fluent is a tuple of the form $\langle v, d \rangle$ where $v \in V$ and $d \in D_v$, which indicates that the variable v takes the value d.

A fluent relates a variable with one of the values in its domain. For instance, let us assume that a planning task features a truck object called `t1` which can be located at three different locations `loc1`, `loc2` or `loc3`. Then, the position of the truck `t1` is a variable named `at-t1` which can take on any value from its domain $D_{\texttt{at-t1}} = \{\texttt{loc1}, \texttt{loc2}, \texttt{loc3}\}$, and $\langle \texttt{at-t1}, \texttt{loc1} \rangle$ is a fluent denoting that truck `t1` is located at the spot `loc1`.

A problem state S is a set of fluents. Consequently, the initial state I and the goal state G of a planning task T are defined through a set of fluents. The set of actions A is also defined in terms of variables and their values.

Definition 2. (Action) An action $a \in A$ is a tuple $\langle PRE(a), EFF(a) \rangle$ where $PRE(a) = \{p_1, \ldots, p_n\}$ is a set of fluents that represents the preconditions of a and $EFF(a)$ is a set of variable assignments of the form $v = d$ that model the effects of a.

Executing an action a in a world state S leads to a new world state S' as a result of applying $EFF(a)$ in S. An effect of the form $v = d'$ assigns the value d' to the variable v; i.e., it adds the fluent $\langle v, d' \rangle$ to state S' and any fluent in S of the form $\langle v, d \rangle$, $d \neq d'$, is removed in S' (to eliminate any fluent that contradicts $\langle v, d' \rangle$).

Partial-Order Planning (POP)

Partial-Order Planning (POP) comes up in the early 90's as an approach to overcome the limitations of state-based planners, mainly the restrictive linear ordering of the plan actions (total-order). The basic idea of POP is that an action a_i is ordered wrt another action a_j if a_i is needed to satisfy a precondition of a_j, or viceversa, or when a conflict appears between them. In any other case, no ordering is established between a_i and a_j, thus avoiding an early commitment on the ordering of the actions [27].

POP operates on partial-order plans. The two key POP operations are introducing an action to satisfy the precondition of another action in the plan and solving a conflict between actions of the plan. Initially, a POP procedure starts from a goal (fluent) $g \in G$ and finds an action a that *supports* or satisfies g; that is, it finds an action a such that $g \in EFF(a)$. In turn, the fluents in $PRE(a)$ must also be satisfied by finding an action of the plan, or introducing a new action, which effects support these fluents. As long as actions are introduced in the plan, negative interactions may arise as a consequence of conflicting preconditions and effects of the actions. Supporting a precondition of an action requires to insert a *causal link*, and solving a conflict involves checking the existence of *threats* [14].

Definition 3. (Causal link) A causal link is a relation between two actions, a_i and a_j, represented by $a_i \xrightarrow{\langle v,d \rangle} a_j$, meaning that the precondition $\langle v, d \rangle$ of a_j is supported by an effect $v = d$ of a_i. a_i is said to be the producer action and a_j the consumer action.

A causal link between a_i and a_j implicitly establishes an ordering between both actions as the producer action a_i must be ordered before the consumer a_j.

Definition 4. (Threat) A threat represents a conflict between an action of the plan and a causal link. An action a_k causes a threat over a causal link $a_i \xrightarrow{\langle v,d \rangle} a_j$ if $v = d' \in EFF(a_k)$ and $d \neq d'$, and a_k is unordered with respect to a_i and a_j. Then, it is said that a_k threatens the causal link $a_i \xrightarrow{\langle v,d \rangle} a_j$.

A threat can be solved by *promoting* or *demoting* the threatening action with respect to the causal link. Specifically, promotion implies introducing an ordering constraint of the form $a_k \prec a_i$, and demotion implies introducing the ordering constraint $a_j \prec a_k$.

Definition 5. (Partial-order plan) A partial-order plan is a tuple $\Pi = \langle \Delta, OR, CL \rangle$. $\Delta \subseteq A$ is the set of actions in Π. OR is a set of ordering constraints ($\prec$) on Δ and CL is a set of causal links over Δ.

This definition of a partial-order plan represents the mapping of a plan into a directed acyclic graph, where Δ represents the nodes of the graph (actions) and OR

and CL are the sets of directed edges that describe the precedences and causal links among these actions, respectively.

A partial-order plan $\Pi = \langle \Delta, OR, CL \rangle$ is a *solution plan* if the preconditions of all the actions in Δ are supported, i.e., there exists a causal link for each precondition, and Π does not contain any threats.

Unlike state-based planners, where the nodes in the search tree represent problem states, the nodes of a POP search tree are partial-order plans. The root node of a POP tree is the minimal initial plan, which contains two fictitious actions: the initial action a_{init}, with no preconditions and $EFF(a_{init}) = I$, and the goal action a_{goal}, with no effects and $PRE(a_{goal}) = G$. In the initial plan of the root node there is only one ordering relation, $a_{init} \prec a_{goal}$. The POP search algorithm works by following these four steps: 1) select a node Π of the tree; 2) select one precondition of an action in Π that is not supported yet (subgoal); 3) choose an action to support the selected subgoal and introduce the corresponding causal link (the selected action can be one that already exists in Π or a newly inserted action in Π); and 4) solve the threats that arise in Π as a consequence of the new data. These four operations are repeated until a solution plan (a node of the tree) is found.

POP performs a plan-based, backward search process, refining partial plans through the addition of actions, causal links and ordering constraints. POP is based on the least commitment strategy [35], which defers planning decisions during the search process and introduces partial-order relations among actions rather than enforcing a concrete order among them.

Forward-chaining in POP

Forward-chaining in POP is aimed at preserving the benefits of partial-order plan construction within the forward-search framework. Like traditional partial-order planners, each node in the search tree represents a partial-order plan. Forward-chaining POP is motivated by the observation that the forward search approach can be seen as committing to a sequence of choices of actions, but not necessarily to the order of their application. By reducing the ordering constraints that are imposed during the construction of the sequence of action choices we retain elements of the least-commitment approach and are able to produce partially-ordered plans with the robustness and flexibility that they can offer.

MAPL [4] and POPF [7] are the first planners that follow this approach. They select a node of the search tree, which is a partial-order plan, and generate a successor node for each new action that can be inserted in the selected plan. When an action is inserted in the plan, these planners only seek to introduce the ordering constraints needed to resolve threats, rather than enforcing an ordering between the new action and all the actions that are already in the plan. The mechanism that MAPL and POPF use to find the actions that can be inserted in a plan Π is as follows: they infer a *state* from the plan Π and check the actions whose preconditions hold in such a state. This state is called *frontier state*.

Definition 6. (Frontier state) The frontier state S_Π of a partial-order plan $\Pi = \langle \Delta, OR, CL \rangle$ is the set of fluents $\langle v, d \rangle$ achieved in Π by an action $a \in \Delta / (v = d) \in EFF(a)$, such that any action $a' \in \Delta$ that modifies the value of v ($(v = d') \in EFF(a'), d \neq d'$) is not reachable from a by following the orderings and causal links in Π.

Landmarks

A landmark is defined as a fluent that must be true at some point during the execution of *any solution plan* [18]. A landmark denotes an indispensable information that needs to be achieved in every solution plan, like the initial and goal fluents, which are trivial landmarks. From this point on, more indispensable information can be deduced through a process aimed at discovering new more landmarks and landmark ordering constraints. For the purpose of this work, we will use the extraction method as explained in [18] based on the relaxation of a planning task [3].

Planning tasks usually have inherent constraints concerning the best order in which to achieve the goals. The extraction and analysis of landmarks has proved to be very helpful to discover such constraints and use them for guiding search. For this reason, many planning systems, which follow different planning paradigms, use landmark-based heuristics [28].

Planning algorithm of FLAP

FLAP follows a forward-chaining POP approach but, unlike MAPL and POPF, FLAP does not require to compute the frontier state of a plan to determine the actions that can be inserted in a plan (generation of the successor nodes). In FLAP, frontier states are only used for evaluation purposes and not for determining the expansion of a node in the POP tree.

The search in FLAP starts with an initial plan $\Pi_0 = \langle\{a_{init}\}, \emptyset, \emptyset\rangle$. Although Π_0 does not contain the fictitious goal action a_{goal}, this action is available to be added to the plan as the rest of actions in A, i.e. $a_{goal} \in A$. In fact, a solution plan is found when a_{goal} is inserted in the plan.

A node of the POP tree represents a partial-order plan; the root node is the plan Π_0. Given a node Π_i, FLAP expands a node Π_i by adding all possible actions that can be supported with the actions of Π_i and solving the corresponding threats. This way, we consider that Π_j is a successor of Π_i if the following conditions are met:

- Π_j adds a new action a_j to Π_j, i.e., $\Delta_j = \Delta_i \cup \{a_j\}$
- All preconditions of a_j are supported with actions of Π_i by inserting the corresponding causal links: $\exists a_i \xrightarrow{p} a_j \in CL_j, a_i \in \Delta_i, \forall p \in PRE(a_j)$.
- All threats in Π_j are solved through promotion or demotion by adding new ordering constraints; the result is that Π_j is a conflict-free plan, that is, a plan with no threats.

FLAP works similarly to a classical POP algorithm: it supports the preconditions of the new action a_j through causal links and solves the threats originated by a_j through ordering constraints. FLAP works differently to a classical partial-order planner since it performs forward-chaining search instead of backward reasoning. Particularly, when expanding a node Π_i, FLAP generates a successor node Π_j for each possible combination of supporting the preconditions of a_j with the actions in Π_i. Note that FLAP only creates Π_j if all preconditions of a_j are solved with the actions in Π_i and the added causal links do not provoke a threat in Π_j. Hence, a node in FLAP always represents a threat-free partial-order plan.

FLAP applies an A* search by using the standard function $f(n) = g(n) + h(n)$ [30]. From the set of open nodes, which initially only contains the plan Π_0, we select the best node according to the evaluation of $f(n)$, where g is the cost to reach the node n measured as the number of actions of the plan in n, and h is the heuristic estimate to reach the goal from n. Once a node Π is selected for expansion, all possible successors of Π are generated, evaluated and added to the list of open nodes.

The planning algorithm of FLAP is sound and complete since it generates all the successors of every expanded node. This way, when a_{goal} is added to the plan, all the goals of the planning task are supported and the plan consistency is guaranteed. Unlike classical POP algorithms, the search process in FLAP may generate many repeated plans. This is a common problem that appears in most forward-search planners and that FLAP avoids by applying a memoization technique.

Heuristic evaluation

One of the strengths of a forward state-space search over a regressive plan-space search is that state-based heuristics are more informed than classical POP-based heuristics. Given a partial-order plan Π, FLAP applies a combination of state-based heuristics in the frontier state S_Π.

The heuristic evaluation in FLAP is carried out by using two different heuristic functions that also incorporate information obtained from a landmarks extraction of the planning task [18]. The following subsections provide a thorough explanation on the heuristic evaluation.

h_{DTG}: a DTG-based heuristic function

A Domain Transition Graph (DTG) of a state variable is a representation of the ways in which the variable can change its value [15]. The transitions of the graph are labelled with the conditions that are necessary for a variable change its value. These conditions are the common preconditions to all the actions that induce the transition. DTGs are independent of the frontier state of the plan, which avoids the need of calculating a DTG in each node of the search tree. This implies that a heuristic based on the information of the DTGs, h_{DTG}, is less costly to compute than the traditional FF heuristic (h_{FF}) [17], which requires the construction of a relaxed planning graph to evaluate each search node.

Algorithm 1 details the procedure for computing the h_{DTG} value of a given frontier state S. $h_{DTG}(S_\Pi)$ is an estimate of the number of actions of a relaxed plan to reach the goals G of the planning task from the frontier state S_Π.

h_{DTG} performs a backward search from G consecutively introducing actions in the relaxed plan until the preconditions of all actions are supported. The procedure for building the relaxed plan handles a list of fluents, *openGoals*, initially set to G. The process iteratively extracts a fluent from *openGoals* and supports it through the insertion of an action in the relaxed plan. The preconditions of such action are then included in the *openGoals* list, and so on.

For each variable $v \in V$, we use a list of values, $Values_v$, which is initialized to the value of v in the frontier state S. For each action inserted in the relaxed

Algorithm 1 h_{DTG} heuristic calculation of a frontier state S

$h_{DTG} \leftarrow 0$
$openGoals \leftarrow G$
for all $\langle v,d\rangle \in G$ **do**
  $Values_v \leftarrow \{d' / \langle v,d'\rangle \in S\}$
end for
while $openGoals \neq \emptyset$ **do**
  $\langle v,d_{end}\rangle \leftarrow \underset{\langle v',d'\rangle \in openGoals}{\arg\max} \; distMin(v', Values_{v'}, d')$
  $openGoals \leftarrow openGoals \setminus \{\langle v,d_{end}\rangle\}$
  $d_{ini} \leftarrow \underset{d \in Values_v}{\arg\min} |Dijkstra(v,d,d_{end})|$
  $minPath \leftarrow Dijkstra(v,d_{ini},d_{end})$
  for $i \leftarrow 0$ to $|minPath|-1$ **do**
    $a_{min} \leftarrow getMinCostAction(v,d_i,d_{i+1})/(d_i,d_{i+1}) \in minPath$
    $openGoals \leftarrow openGoals \cup \{\langle v',d'\rangle \in PRE(a_{min})/d' \notin Values_{v'}\}$
    for all $(v',d') \in EFF(a_{min})$ **do**
      $Values_{v'} \leftarrow Values_{v'} \cup \{d'\}$
    end for
    $h_{DTG} \leftarrow h_{DTG} + 1$
  end for
end while
return h_{DTG}

plan that has an effect $\langle v,d'\rangle$, d' is stored in $Values_v$, meaning that variable v achieves the value d' in the relaxed plan. Hence, the values in $Values_v$ can be used to support the preconditions of the actions that will be inserted in the relaxed plan in the next iterations. Given a frontier state S, h_{DTG} is computed in two stages:

- **Open goal selection**: At this stage, we select a fluent $\langle v,d_{end}\rangle$ from $openGoals$. Similarly to h_{FF}, which selects a subgoal in the last fact layer of the relaxed planning graph, h_{DTG} also selects the most costly fluent in first place. h_{DTG} evaluates the cost of a fluent through the length of the shortest path, $minPath$, of a variable value modification in its DTG, using the classical Dijkstra algorithm (see Algorithm 2).
- **Relaxed plan construction**: $minPath$ is the number of value transitions for v to change its value from d_{ini} to d_{end}, that is, $minPath = \langle (d_{ini},d_i), (d_i,d_{i+1}), \ldots, (d_{i+n},d_{end})\rangle$. For each value transition $(d_i,d_{i+1}) \in minPath$, the minimum-cost action a_{min} that generates the value transition is introduced in the relaxed plan; that is, $\langle v,d_i\rangle \in PRE(a_{min})$ and $\langle v,d_{i+1}\rangle \in EFF(a_{min})$. The cost of an action is estimated as the sum of the cost of achieving all its preconditions, as shown in Algorithm 3.
  The unsupported preconditions of every a_{min} inserted in the relaxed plan are stored in $openGoals$, so they will be supported in forthcoming iterations. For each effect $\langle v',d'\rangle \in EFF(a_{min})$, the value d' is stored in $Values_{v'}$, and thus d' can be used in the following iterations to support the subsequent fluents of $openGoals$.

Algorithm 2 $distMin(v, Values_v, d)$ function

$d_{ini} \leftarrow \underset{d' \in Values_v}{\arg\min} |Dijkstra(v,d',d)|$
return $|Dijkstra(v,d_{ini},d)|$

Algorithm 3 $getMinCostAction(v,d_i,d_j)$ function

$A_{ij} \leftarrow a \in A / \langle v,d_i\rangle \in PRE(a) \wedge \langle v,d_j\rangle \in EFF(a)$
$a_{min} \leftarrow \underset{a \in A_{ij}}{\arg\min} \sum_{\forall \langle v',d'\rangle \in PRE(a)} distMin(v', Values_{v'}, d')$
return a_{min}

The iterative evaluation procedure goes on until all the open goals have been supported, that is, $openGoals = \emptyset$. When this occurs, h_{DTG} returns the number of actions in the relaxed plan.

h_{FF}: FF heuristic function

FLAP also makes use of the traditional FF heuristic function h_{FF} [17], which builds a relaxed plan by ignoring the delete effects of the actions. $h_{FF}(S_\Pi)$ returns, as well as $h_{DTG}(S_\Pi)$, an estimate of the number of actions necessary to reach the goal state G from S_Π.

The calculation of h_{FF} is most costly than h_{DTG} because h_{FF} needs to build a relaxed planning graph in each node Π of the search tree. There is no an efficient way to compute this graph in an incremental way because it would require to propagate the changes in the frontier state S_Π across the graph. However, building a new relaxed planning graph at each node provides an updated heuristic information and, therefore, h_{FF} often offers more accurate evaluations than h_{DTG}.

h_{LAND}: Landmarks heuristic

Landmarks are fluents that must be achieved in every solution plan [18,31]. Like the LAMA planner [28], FLAP computes a landmark graph (considering only necessary and reasonable orderings) and uses this information to calculate heuristic estimates. Since all landmarks must be achieved in order to solve the planning task, the value of $h_{LAND}(\Pi)$ can be estimated through the set of landmarks that still need to be achieved to reach the goal state G from the frontier state S_Π.

A plan Π can be seen as a sequence of states that are traversed to reach the frontier state S_Π from the initial state I. Then, we consider that a landmark l is *accepted* in Π if it holds in one of these states and all landmarks ordered before l have been already accepted in that state. Once a landmark is accepted, it remains accepted in all successor states. When the set of non-accepted landmarks is calculated, $h_{LAND}(\Pi)$ is the result of estimating the cost of reaching these landmarks with either h_{DTG} or h_{FF}. Hence, we have two versions

of the landmarks heuristic, which we call h_{LAND_DTG} and h_{LAND_FF}, respectively.

Combination of heuristic functions

For evaluating a plan $\Pi = \langle \Delta, OR, CL \rangle$, FLAP defines two different evaluation functions:

- $f_{FF}(\Pi) = w_1 * g(\Pi) + w_2 * h_{LAND\_FF}(\Pi) + w_3 * h_{FF}(S_\Pi)$
- $f_{DTG}(\Pi) = w_1 * g(\Pi) + w_2 * h_{LAND\_DTG}(\Pi) + w_3 * h_{DTG}(S_\Pi)$

$g(\Pi)$ measures the cost of Π as the number of actions in Π; i.e., $g(\Pi) = |\Delta|$. We learned the values $w_1 = 1$, $w_2 = 4$ and $w_3 = 2$ by a trial-and-error empirical evaluation over the planning benchmark problems (see the section *Experimental results*). Heuristics functions are given more weight than the cost function to obtain a greedy-like search. Additionally, the weight of the landmarks heuristic is twice as much as the weight of h_{FF} or h_{DTG} because the values of h_{LAND} are typically lower than the estimates of h_{FF} or h_{DTG}, particularly in problems where the number of extracted landmarks is rather small.

FLAP uses both evaluation functions, f_{FF} and f_{DTG}, to simultaneously explore different parts of the search space and hence have more chances to escape from plateaus, as it is described in the following subsection.

Parallel searches for plateau escaping

In heuristic planning, search nodes are often inaccurately evaluated and the search may be misled into large local minima/plateaus, thus resulting in a performance degradation. The problem of escaping from plateaus has been addressed in different ways:

- Several approaches use greedy best-first search to avoid plateaus. They tackle this issue by adding a *diversity* to search [19,22], which is an ability in simultaneously exploring different parts of the search space to bypass large errors in heuristic functions.
- A different strategy lies in combining/alternating different heuristics (or search parameters) to diversify the search directions [29,33].
- Other approaches adding a diversity with an application to planning include a restarting procedure combined with local search [9] and random walk [25].

In FLAP, we apply a new strategy for plateau escaping based on the ideas proposed in the aforementioned works. The main A* search starts from the initial empty plan, Π_0, by using the f_{FF} evaluation function. Although f_{DTG} is actually faster to compute than

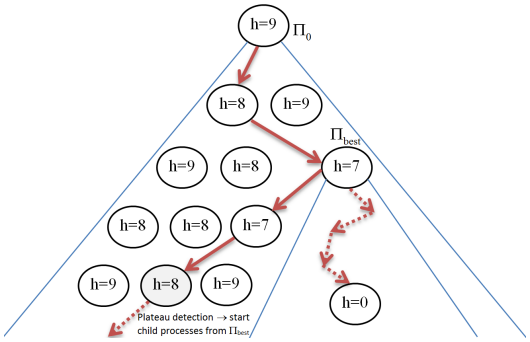

Fig. 1. Parallel A* child search started when the main search gets stuck in a plateau.

f_{FF}, our current implementation of h_{DTG} is not as robust as h_{FF} (see section *Limitations and extensions of FLAP*).

For any A* search in FLAP, the plan with the best heuristic value reached so far, Π_{best}, is stored. For the main A* search, Π_{best} is initially set to the initial empty plan, Π_0, but, when another plan is found with a strictly better heuristic value, Π_{best} is set to that plan. We consider the main A* search is stuck in a plateau when Π_{best} is not updated in several iterations. In this case, two new A* child searches are started in parallel from Π_{best}, as it can be observed in Figure 1. One process uses f_{DTG} whilst the other one uses f_{FF}, in order to diversify the search in two different directions.

The goal of a child search is not to escape from the plateau, but to find a solution plan from the frontier state of Π_{best}, which is likely to be closer to the goal.

If a child search is successful in finding an exit to the plateau of its parent search, it will continue searching for a solution plan. If this child search is stuck in a plateau again, it repeats the same diversification process and starts its own two parallel searches to speedup the progress towards a solution.

Since any search can potentially start two new child processes, it is necessary to control the possible exponential growth in the number of parallel search processes. Then, a search process is terminated when one of the following circumstances occur:

- A solution plan is found. The current version of FLAP stops when a solution is found instead of finding more solutions.
- If a search manages to exit from a plateau, i.e. its Π_{best} is updated, then all its descendant search processes are cancelled. The only exception to this rule applies to the child search which has the best global heuristic value.

In practice, the number of simultaneous search processes does not usually exceed the number of processing cores (8 in our test computer) in the tested problems.

Comparison between FLAP and OPTIC

OPTIC, the most recent version of POPF, is a forward-chaining POP that also handles temporal planning problems, soft constraints and preferences [1]. In this section we compare the planning algorithms of OPTIC and FLAP and we show the characteristics that make FLAP be more flexible than OPTIC.

There are three main differences between FLAP and OPTIC. In the first place, unlike our parallel A*-search algorithm, OPTIC uses the same enforced hill-climbing (EHC) algorithm as FF [17]. EHC finds a first plan very quickly but it may often yield low-quality solutions. When the EHC search fails, OPTIC switches to a best-first search to ensure completeness.

In the second place, regarding the heuristic evaluation, OPTIC uses an extended version of h_{FF} while FLAP uses the combination of heuristics explained in the previous section.

The third difference between the two planners relies on the generation of the successor nodes. This is the key feature that allows FLAP to fully exploit the least-commitment principle of POP. Given a partial-order plan Π, OPTIC only considers the actions whose preconditions hold in the frontier state S_Π whereas FLAP considers the actions whose preconditions are supported with the actions in Π. The approach of OPTIC is a simple and straightforward process for the following reasons:

- Checking whether the preconditions of an action hold in a state is processed very quickly.
- No threats arise when the new action is added to Π because this action is inserted after all the actions of Π which the new action may have conflicts with.
- The number of actions that can be supported in S_Π is usually smaller than the number of actions that can be added throughout Π, thus leading to a smaller branching factor during the search.

Obviously, expanding a node in FLAP is a more costly operation since any action in Π is a potential support for the preconditions of the new action and, additionally, it is necessary to fix the threats caused by new action. However, the approach of FLAP has two advantages; it reduces the need of backtracking, and it improves the plan parallelism.

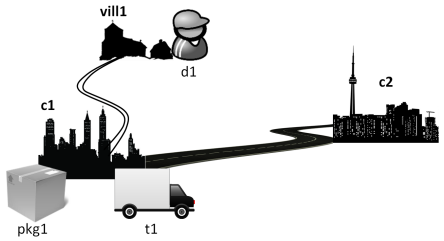

Fig. 2. Scenario example of the *DriverLog* domain.

In order to illustrate the advantages of FLAP, we use a particular example of the *DriverLog* domain represented in Figure 2. In this domain, presented in the 2002 International Planning Competition (IPC) [11,24], a collection of trucks is used to deliver some packages to their destination cities, and trucks require a driver to move.

The goal of this problem is to have package `pkg1` at city `c2`, i.e. $G = \{\langle$`at-pkg1, c2`$\rangle\}$. Let us suppose that both planners, OPTIC and FLAP, reach a search node that contains the plan shown in Figure 3. This plan consists of three sequential actions: 1) the driver `d1` walks from the village `vill1` to city `c1`, 2) `d1` gets on truck `t1` and 3) `d1` moves `t1` to `c2`. As it can be observed, these three actions are necessary to solve the problem.

In order to reach the goal, it is necessary to insert the action `load pkg1 t1 c1`, which we will call a_{new}, to load `pkg1` on `t1` in city `c1`. The frontier state of the plan in Figure 3 does not support the preconditions of a_{new} because the truck is no longer in `c1`. This way, OPTIC needs to take the truck back to `c1`, reaching a repeated frontier state which is pruned by the memoization mechanism. Then, OPTIC's solution is to backtrack to a node of the search tree in which a_{new} is applicable.

FLAP, however, seeks producer actions in the plan that support the preconditions of a_{new}, and it finds one way of doing this as shown in Figure 4. Specifically:

- The preconditions of a_{new} are supported by the existing actions in the plan. In this case, its two preconditions $\langle$`at-pkg1,c1`$\rangle$ and $\langle$`at-t1,c1`$\rangle$ are produced by the initial fictitious action, a_{init}.
- A conflict appears with the action `drive t1 c1 c2 d1` of the plan, since this action threatens the causal link $a_{init} \xrightarrow{\langle at-t1,c1\rangle} a_{new}$ This threat is solved by demotion through the insertion of the ordering constraint $a_{new} \prec$ `drive t1 c1 c2 d1`.

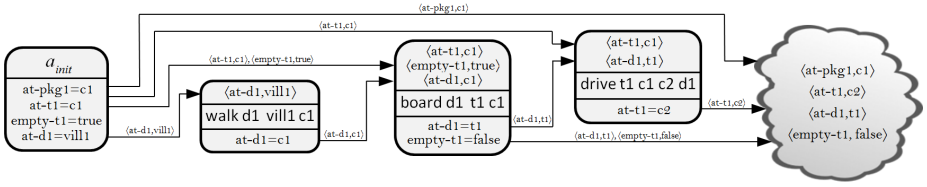

Fig. 3. Partial plan computed for the *DriverLog* problem example. For each action, its preconditions are shown at the top of its box and its effects at the bottom. The frontier state of the plan is displayed on the right.

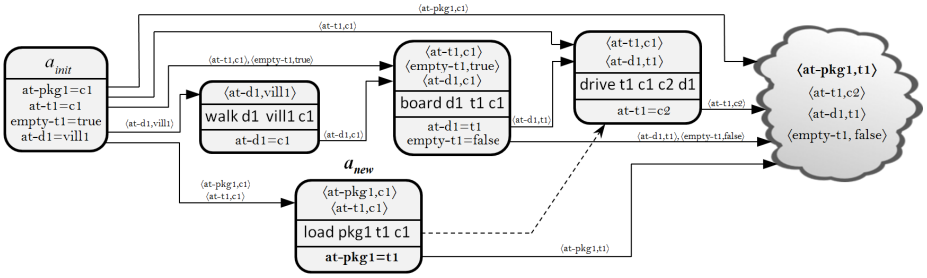

Fig. 4. Partial plan computed for the *DriverLog* problem example. Solid lines represent causal links and dashed lines represent orderings.

– The resulting successor plan, shown in Figure 4, is not a repeated node because the frontier state of this plan contains the new fluent ⟨at-pkg1,t1⟩.

Finding actions that support the preconditions of a new action all along the plan requires an extra computational cost but, on the other hand, it avoids backtracking since the new generated plan (Figure 4) can be easily extended to reach a solution plan. In simple problems, where no backtracking is often required, FLAP is usually slower due to the additional calculations but, in hard problems, this approach allows FLAP to significantly outperform OPTIC.

The second advantage of the approach followed by FLAP is that it can generate plans with a smaller makespan (plan duration) than OPTIC. To illustrate this, we show in Figure 5 the solution plans provided by OPTIC and FLAP for the second problem of the *Rovers* domain used in the IPC 2006. In this domain, a collection of rovers with different equipment navigate a planet surface to collect and analyse samples of soil and rock, take pictures of diverse objectives and communicate the results to a lander.

As it can be observed, both plans have the same eight actions. However, the plan produced by OPTIC has a makespan of 5 time steps, one more than the plan of FLAP. This is because, in OPTIC, the order in which the actions are inserted in the plan is determinant: action communicate_rock_data is inserted when action communicate_image_data was already in the plan, hence ordering the first one after the second one (communications cannot be done in parallel). In contrast, the insertion order is not important in FLAP as it does not prevent it to get the best-quality solutions.

In summary, we can conclude that the forward-chaining POP approach of FLAP is more flexible than the one used in OPTIC since FLAP does not restrict itself to only inserting actions in the frontier state. More specifically, the late-commitment approach of OPTIC is based on delaying commitment to ordering decisions on the frontier state, which makes it lose flexibility as it ignores the possibility of exploiting white knights [5] or considering the demotion strategy for an action that interferes with a fluent in the frontier state.

Limitations and extensions of FLAP

FLAP was designed as a general and flexible planner, which can easily been extended to handle problems with complex features. Our next step is to implement a new version of FLAP to deal with temporal planning problems. In these type of problems, actions may have different durations, their preconditions may be required to hold during the whole action duration, and their effects can also occur at the beginning of the action.

One of the limitations of the current version of FLAP is that the goal distance is estimated as the number of actions required to reach the goals: both h_{FF} and h_{DTG} return the number of actions in the computed

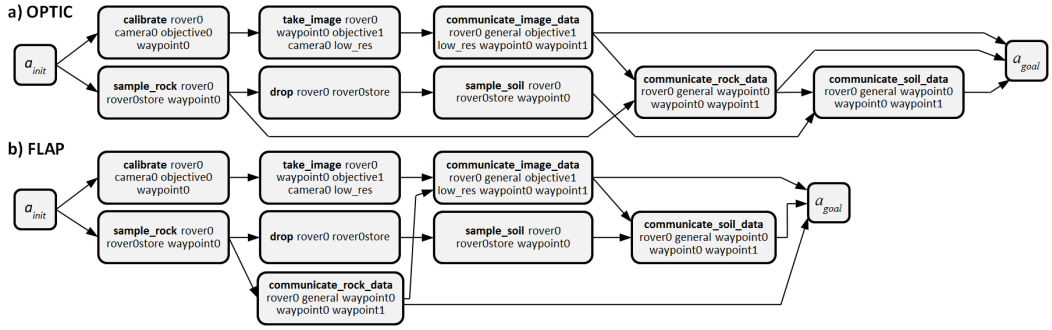

Fig. 5. Solution plan computed by a) OPTIC and b) FLAP for the second problem file of the *Rovers* domain.

relaxed plans. This is a valid approach to guide the search towards a solution but makes it more difficult to find good-quality plans regarding the plan duration or makespan, which is the main metric used in temporal planning. Then, a first modification to handle temporal problems is to adjust the heuristic functions to estimate the makespan, returning the duration of the obtained relaxed plans instead of the number of actions. This change does not imply a significant increase in the computational cost of these heuristics, as it only requires to record the time points in which the effects are achieved.

Another limitation is that h_{DTG} does not identify dead-end states properly, returning values different from ∞ in these cases. The reason is that h_{DTG} is only able to detect a dead-end state if the DTG of a variable does not contain a transition path from the current value of the variable to its goal value. Then, h_{DTG} does not consider the interactions between different variables to detect dead-ends. This problem can be mitigated by computing mutex fluents in a preprocessing stage since they can be used to detect conflictive interactions among fluents.

Finally, an additional extension of FLAP is to consider the use of a portfolio approach [26]. This approach is based on the following idea: several planning algorithms are executed in sequence with shorter timeouts, expecting that at least one of them will find a solution in its allotted time. FLAP combines several techniques and heuristic functions, and none of them clearly dominates the other ones, so a portfolio approach could be easily implemented to check if it reports a significative improvement in the planning performance.

Experimental results

In this section, we address two issues. On the one hand, we look into the reasons that led us to use f_{FF} as the evaluation function for the main search in FLAP. For this purpose, we compared the performance of f_{FF} and f_{DTG} to show the advantages and drawbacks of both functions. This comparison is presented in the following subsection.

On the other hand, we also present some experimental results to show the performance of FLAP in terms of plan quality and computational time. For this purpose, we compared FLAP with three well-known planners that return plans with parallel actions: OPTIC, LPG-td and Temporal Fast Downward (TFD). A brief description of these planners and the results of this comparison are discussed in the second subsection.

We selected ten propositional domains from the International Planning Competitions (IPC) [24,11]. The IPCs provide a wide set of benchmarking problems to assess the performance of the planners [23]. The tested domains are described below:

- *Blocksworld*: this domain, from the 2000 IPC, consists of a set of blocks that must be arranged to form one or more towers. We have used a variation of this domain where several robot arms are used to handle the blocks, thus allowing parallel actions in the plans.
- *Depots*: this domain, introduced in the 2002 IPC, combines a transportation-style problem with the *Blocksworld* domain.
- *Driverlog*: this domain, used in the 2002 IPC, involves transportation, but vehicles need a driver before they can move.
- *Elevators*: in this domain, used in the 2011 IPC, several elevators of different types must transport several passengers to their floors.
- *Logistics*: in this domain, introduced in the IPC 2000, several airplanes and trucks cooperate to deliver some packages to their destinations.
- *Openstacks*: in this domain, from the 2011 IPC, a manufacturer has a number of orders to produce, each one consisting of a combination of products.

– *Satellite*: this domain, used in the 2004 IPC, involves satellites collecting and storing data using different instruments to observe a set of targets.
– *Rovers*: used in the 2006 IPC, the objective is to use a collection of mobile rovers to traverse between waypoints on the planet, carrying out a variety of data-collection missions and transmitting data back to a lander.
– *Woodworking*: used in the 2011 IPC, the goal is to manufacture some wood pieces by using a set of different machines in a production chain.
– *Zenotravel*: in this domain, presented in the 2002 IPC, people must embark onto planes, fly between locations and then debark, with planes consuming fuel at different rates according to their speed of travel.

Testing was performed on a 2.3 GHz i7 computer with 16 GB of memory running Ubuntu 64-bits.

Comparison of the evaluation functions

As we have described before, FLAP uses two evaluation functions, f_{FF} and f_{DTG}, to guide the search processes. f_{FF} uses the classical heuristic of the FF planner, while f_{DTG} uses a new heuristic based on the computation of shortest paths in the DTGs of the variables. To compare the performance of both functions, we developed a simple version of FLAP that only uses a single A* search (no parallel search). First, we used f_{FF} to guide the search and, then, we used f_{DTG} to check the differences.

Regarding the time performance, measured in expanded nodes per second, the search is always faster with f_{DTG}. One of the advantages of h_{DTG} is that it can be computed several times faster than h_{FF}, as h_{DTG} does not require to build a new graph in each search node. We observed that, for example, h_{FF} takes 0.85 ms. in average to evaluate a plan in a problem with about 5000 ground actions, while h_{DTG} takes only 0.12 ms. However, the speed up of f_{DTG} is more noticeable in problems where the size of the relaxed planning graphs of h_{FF} is rather large. For this reason, we only compared the search performance in the largest problems (the last 7 problems of each domain). The results are depicted in Table 1. The domains in which the speed up of f_{DTG} is less significant are *Blocksworld*, *Satellite* and *Logistics*. On the contrary, the domains in which the performance increase is more noticeable are *Woodworking*, *Openstacks* and *Zenotravel*.

We thus confirmed that the use of f_{DTG} instead of f_{FF} speeds up the search to a greater or lesser extent.

	f_{DTG}		f_{FF}	
Domain	nodes/sec.	exp.nodes	nodes/sec.	exp.nodes
Blocksworld	58.86	241,00	58.35	31,43
Depots	118.89	245,14	101.64	33,14
Driverlog	517.26	97,00	431.84	160,29
Elevators	367.96	32,14	234	30,43
Logistics	72.45	110,86	69.08	115,86
Openstacks	45.11	117,86	21.88	117,71
Rovers	141.4	2576,86	111.82	239,29
Satellite	4.8	144,86	4.2	50,14
Woodworking	158.27	–	52.89	–
Zenotravel	3.9	67,43	2.1	83,86

Table 1

Search speed (in nodes per second) and average number of expanded nodes, using the f_{DTG} or the f_{FF} evaluation function.

Therefore, the decision of using f_{FF} instead of f_{DTG} in the main search of FLAP is only motivated because h_{FF} is a better informed heuristic than h_{DTG}. As it can be observed in Table 1, f_{DTG} only guides the search better, i.e. needs to expand fewer nodes to find a solution, in three out of the ten domains: *Driverlog*, *Logistics* and *Zenotravel*. In these domains, the DTGs of some variables are quite large and sparse (for example, DTGs of the position of the trucks and the drivers in the *Driverlog* domain), which allows us to obtain longer and more informative paths of value transitions.

However, f_{FF} is more accurate in the evaluations of the remaining seven domains. In particular, it significantly outperforms f_{DTG} in the *Woodworking* domain, for which f_{DTG} was not able to obtain any solution. This is a non-reversible domain, which causes that many of the frontier states reached during the search are dead-ends, and h_{DTG} is not able to identify dead-ends properly. The better behaviour of f_{FF} in most of the domains led us to use this evaluation function for the main search of FLAP. Nevertheless, the alternation of both evaluation functions, f_{FF} and f_{DTG}, as a mechanism to rapidly escape from the plateaus is one of the keys of the good performance of FLAP.

Comparison between parallel planners

The goal of this comparison is to demonstrate that FLAP, a planner compliant with the least-commitment principle, can be competitive with other state-of-the-art planners capable of generating parallel plans. For this purpose, we have selected two partial-order planners, LPG-td [13] and Temporal Fast Downward (TFD) [10], aside from OPTIC.

LPG-td is an extended version of the LPG planner [12]. The basic search scheme of LPG was inspired by Walksat, an efficient procedure to solve SAT-problems. The search space of LPG consists of *action graphs*, particular subgraphs of the planning graph representing partial plans. The search steps are certain graph modifications transforming an action graph into another one. We have selected LPG-td for the comparison since action graphs are an alternative way for representing partial-order plans, thus allowing to generate plans with parallel actions. LPG-td is sub-optimal and incomplete, due to its stochastic local-search approach, but it can find a first solution very quickly. We have used LPG-td with a fixed seed of 0 for the random number generator in order to obtain reproducible results.

Temporal Fast Downward (TFD) is a variant of the propositional Fast Downward planning system [16]. TFD uses a greedy best-first search approach enhanced with deferred heuristic evaluation. Besides the values of the state variables, the time-stamped states in the search space contain a real-valued time stamp as well as information about scheduled effects and conditions of currently executed actions. This integrated process of action selection and time scheduling yields very good results in terms of plan quality according to the makespan.

We have run all the benchmark problems from the ten selected domains with these planners. Each experiment was limited to 30 minutes of wall-clock time and to the 16Gb of available memory. For this comparison, we have only considered the first plan returned by the planners.

As it can be observed in Table 2, FLAP is able to solve all tested problems. The problem in which FLAP took longer to find a solution was the last one of the *Zenotravel* domain, and it was solved in 65.82 seconds. LPG-td only fails one problem, concretely the problem 16 of the *DriverLog* domain. On the contrary, OPTIC fails to solve many of the *Depots* and *Blocksworld* problems. In these problems, with a large number of interactions between the goals, the use of landmarks would have helped OPTIC be more effective. It has also difficulties to deal with the latest problems of *Driverlog*, *Satellite* and *Zenotravel*, due to the size of these problems and the complexity of the plateaus that appear (above all in the *Driverlog* domain). TFD also has difficulties with the *Depots* domain and with the latest problems of *DriverLog* and *Rovers*.

As for the plan quality, Table 3 shows the average makespan of the solution plans obtained by these five planners. For computing these values we have only taken into account the problems that the planners were able to solve.

Domain (#problems)	FLAP	OPTIC	LPG-td	TFD
BlocksWorld (34)	34	24	34	34
Depots (20)	20	11	20	10
DriverLog (20)	20	15	19	16
Elevators (30)	30	30	30	30
Logistics (20)	20	20	20	20
OpenStacks (30)	30	30	30	30
Rovers (20)	20	20	20	17
Satellite (20)	20	16	20	20
WoodWorking (30)	30	28	30	30
ZenoTravel (20)	20	16	20	20
Total (244)	244	210	243	227
Coverage	100%	86,07%	99,59%	93,03%

Table 2

Number of problems solved in the tested domains.

TFD is the planner that yield better shorter plans in most of the tested domains. It only produces slightly worse solutions than FLAP in the *Logistics*, *Rovers* and *ZenoTravel* domains. As it can be observed, the quality of FLAP is very similar to the one of OPTIC. These are very promising results as we have to take into account that FLAP is currently designed to optimize the number of actions in the plans. We expect to get a significant improvement in the durations of the plans through the introduction of some modifications in the heuristic functions to optimize the makespan in a future version of FLAP. LPG-td is the planner that performs worse in this regards since it generates a 164% longer plans than TFD and a 152% than FLAP on average.

Domain	FLAP	OPTIC	LPG-td	TFD
BlocksWorld	15,21	20,85	43,01	9,59
Depots	26,80	30,01	29,63	22,41
DriverLog	29,35	28,16	47,25	28,66
Elevators	12,90	14,37	35,54	12,50
Logistics	16,50	18,03	36,39	16,86
OpenStacks	53,87	49,14	55,68	48,72
Rovers	14,05	16,26	23,82	17,06
Satellite	18,50	13,29	16,99	16,31
WoodWorking	6,33	4,12	6,18	5,75
ZenoTravel	11,00	10,69	15,49	11,70
Average	20,45	20,49	31,00	18,96

Table 3

Average makespan of the solution plans for the tested domains.

Regarding the running time, Figure 6 show the used time by these planners to find a first solution in all the

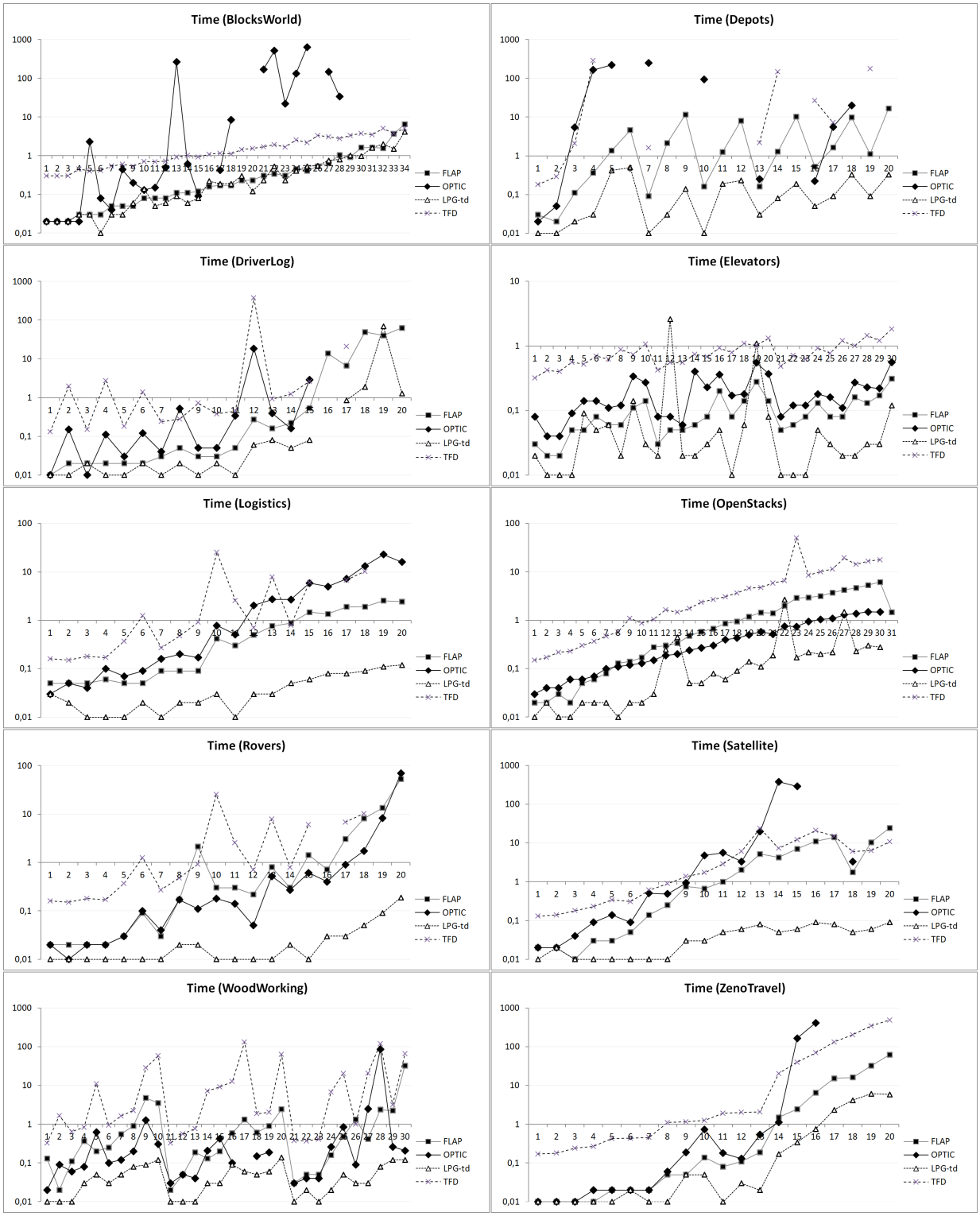

Fig. 6. Search time in second used by FLAP, OPTIC, LPG-td and TFD in the tested domains.

tested problems. As it can be observed, LPG-td is the faster planner. FLAP only obtains a better average time than LPG-td in the *Elevators* domain: 0,1 vs. 0,16 seconds per plan. In the *BlocksWorld* domain the average time is also similar: 0,64 and 0,5 seconds by FLAP and LPG-td, respectively. The local-search approach of LPG-td allows it to find a first solution plan very fast in most of cases, but the quality of these solutions is not very good as Table 3 has shown.

FLAP is faster than OPTIC except on the *OpenStacks* and *Rovers* domains. In these domains the number of explored nodes is smaller in FLAP, so the additional temporal cost is due to the overhead in the computation of the successors of a node: a higher number of actions can be added to the current plan and, moreover, one same action can be inserted in different positions in the plan, so the number of successors of a plan is usually higher in FLAP. Nevertheless, the running times in these domains are quite small and do not prevent FLAP from being about 19 times faster than OPTIC in the tested problems on average.

FLAP is also about 12 times faster than TFD on average. FLAP is able to find solutions in all the tested domains in less time than TFD. On the contrary, the solutions found by TFD are usually shorter, as we have mentioned above.

Aside from the use of a powerful combination of heuristics, one of the key of the performance of FLAP is the mechanism of parallel searches to escape from plateaus. As each search process can launch another two child searches when gets stuck in a plateau, there exists a potential possibility of an exponential growth in the number of parallel processes. However, an as it can be observed in Table 4, the maximum number of simultaneous searches remains at very acceptable levels in all the domains. *Woodworking* is the domain that needs a higher number of parallel searches, mainly due to the lack of accuracy of the h_{DTG} heuristic in these problems. Even so, eight-core processors are very common at present and they can manage five parallel search processes seamlessly.

Finally, we show in Table 4 the memory usage statistics of FLAP for the tested domains. FLAP uses several A* search processes and one of the main drawback of this type of algorithms is the high memory consumption. As it can be observed, the memory consumption is quite restrained since it does not require more than 2GB. in any of the tested problems. This is mainly due to the good heuristic guidance, which helps FLAP find a solution without exploring a large number of nodes.

		Memory usage		
Domain	Threads	Min.	Max.	Average
BlocksWorld	2,18	13	94	22,94
Depots	4,15	14	297	65,10
DriverLog	2,95	12	980	147,40
Elevators	1,13	13	28	14,77
Logistics	2,00	18	104	47,15
OpenStacks	2,07	17	69	35,03
Rovers	3,55	16	566	101,60
Satellite	1,05	13	65	29,10
WoodWorking	5,27	18	1774	128,00
ZenoTravel	1,05	12	118	27,95

Table 4

Average number of maximum parallel search processes (threads) and memory consumption in MB. by FLAP for each domain.

Conclusions

This paper presents FLAP, a hybrid planner that combines partial-order plans with forward search and uses state-based heuristics. FLAP is fully compliant with the least-commitment strategy, avoiding an early commitment to the ordering of the actions. This achieves flexibility, reduces the need of backtracking and produces shorter plans at the expense of a more costly search process. In order to alleviate the search burden, FLAP implements a parallel search technique that diversifies the search when a plateau is found. We also presented an exhaustive analysis to show the differences between the late-commitment approach of OPTIC and the least-commitment approach of FLAP.

We compared FLAP with three modern planners, OPTIC, LPG-td and TFD. Experimental results show that FLAP is able to solve more problems than the other three planners in the tested domains. FLAP also offers a very good trade-off between the quality of the solutions regarding the makespan and the running time. TFD returns the best solutions wrt makespan and FLAP outperforms both OPTIC and LPG-td in this respect. This is a promising result as the current version of FLAP is not focused on optimizing the makespan. With regard to the running time, only LPG-td is faster than FLAP from the tested planners. We can conclude then that FLAP is very competitive when compared with some of the top-performing planners.

As a future extension, we will investigate on the adaptation of FLAP heuristics to optimise the makespan and to mitigate the problem of h_{DTG} with dead-end states in non-reversible domains. We are also working on a temporal version of FLAP, extending the representation to durative actions and adapting the planning algorithm of FLAP to reasoning about time.

Acknowledgements

This work has been partly supported by the Spanish MICINN under projects Consolider Ingenio 2010 CSD2007-00022 and TIN2011-27652-C03-01, the Valencian Prometeo project II/2013/019.

References

[1] J. Benton, A. Coles, and A. Coles. Temporal planning with preferences and time-dependent continuous costs. *International Conference on Automated Planning and Scheduling*, pages 2–10, 2012.

[2] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[3] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[4] M. Brenner. Multiagent planning with partially ordered temporal plans. *Technical Report. Institut für Informatik, Universität Freiburg*, 2003.

[5] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[6] Y. Chen, B. Wah, and C. Hsu. Temporal planning using subgoal partitioning and resolution in SGPlan. *Journal of Artificial Intelligence Research*, 26:323–369, 2006.

[7] A. Coles, A. Coles, M. Fox, and D. Long. Forward-chaining partial-order planning. *International Conference on Automated Planning and Scheduling*, pages 42–49, 2010.

[8] A. Coles, M. Fox, K. Halsey, D. Long, and A. Smith. Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence*, 173(1):1–44, 2009.

[9] A. Coles, M. Fox, and A. Smith. A new local-search algorithm for forward-chaining planning. *International Conference on Automated Planning and Scheduling*, pages 89–96, 2007.

[10] P. Eyerich, R. Mattmüller, and G. Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. *International Conference on Automated Planning and Scheduling*, 2009.

[11] A. Gerevini, P. Haslum, D. Long, A. Saetti, and Y. Dimopoulos. Deterministic planning in the fifth International Planning Competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6):619–668, 2009.

[12] A. Gerevini, A. Saetti, and I. Serina. Planning through stochastic local search and temporal action graphs in LPG. *Journal of Artificial Intelligence Research*, 20:239–290, 2003.

[13] A. Gerevini, A. Saetti, and I. Serina. Temporal planning with problems requiring concurrency through action graphs and local search. *International Conference on Automated Planning and Scheduling*, pages 226–229, 2010.

[14] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning. Theory and Practice*. Morgan Kaufmann, 2004.

[15] M. Helmert. A planning heuristic based on causal graph analysis. *International Conference on Automated Planning and Scheduling*, pages 161–170, 2004.

[16] M. Helmert. The Fast Downward planning system. *Journal of Articial Intelligence Research*, 26:191–246, 2006.

[17] J. Hoffman and B. Nebel. The FF planning system: Fast planning generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[18] J. Hoffmann, J. Porteous, and L. Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

[19] T. Imai and A. Kishimoto. A novel technique for avoiding plateaus of greedy best-first search in satisficing planning. *AAAI Conference on Artificial Intelligence*, pages 985–991, 2011.

[20] D. Kovacs. Complete BNF description of PDDL3.1. Technical report, 2011.

[21] J. Kvarnström. Planning for loosely coupled agents using partial order forward-chaining. *International Conference on Automated Planning and Scheduling*, pages 138–145, 2011.

[22] C. Linares and D. Borrajo. Adding diversity to classical heuristic planning. *Third Annual Symposium on Combinatorial Search (SoCS-10)*, pages 73–80, 2010.

[23] C. Linares, S. Jiménez, and M. Helmert. Automating the evaluation of planning systems. *AI Communications*, 26(4):331–354, 2013.

[24] D. Long and M. Fox. The 3rd International Planning Competition: results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

[25] H. Nakhost and M. Müller. Monte-carlo exploration for deterministic planning. *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 1766–1771, 2009.

[26] S. Núñez, D. Borrajo, and C. Linares. Performance analysis of planning portfolios. *Fifth Annual Symposium on Combinatorial Search (SOCS)*, pages 65–71, 2012.

[27] J. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. *International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–114, 1992.

[28] S. Richter and M. Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 29(1):127–177, 2010.

[29] G. Röger and M. Helmert. The more, the merrier: Combining heuristic estimators for satisficing planning. *International Conference on Automated Planning and Scheduling*, pages 246–249, 2010.

[30] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

[31] L. Sebastia, E. Onaindía, and E. Marzal. Decomposition of planning problems. *AI Communications*, 19(1):49–81, 2006.

[32] A. Torreño, E. Onaindía, and O. Sapena. An approach to multi-agent planning with incomplete information. *European Conference on Artificial Intelligence (ECAI)*, 242:762–767, 2012.

[33] R. Valenzano, N. Sturtevant, J. Schaeffer, K. Buro, and A. Kishimoto. Simultaneously searching with multiple settings: An alternative to parameter tuning for suboptimal single-agent search algorithms. *International Conference on Automated Planning and Scheduling*, pages 177–184, 2010.

[34] M. Veloso and P. Stone. FLECS: Planning with a flexible commitment strategy. *Journal of Artificial Intelligence Research*, 3:25–52, 1995.

[35] D. Weld. An introduction to least commitment planning. *AI magazine*, 15(4):27, 1994.