Minimizing the number of actions in parallel planning

Stéphane Grandcolas, Cyril Pain-Barre

Minimizing the number of actions in parallel planning

Stéphane Grandcolas and Cyril Pain-Barre
LSIS – UMR CNRS 6168
Domaine Universitaire de Saint-Jérôme
Avenue Escadrille Normandie-Niemen
F-13397 Marseille Cedex 20 - France

Email: {stephane.grandcolas,cyril.pain-barre}@lsis.org

Abstract

In the domain of classical planning one distinguishes plans which are optimal in their number of actions, they are referred as sequential plans, from plans which are optimal in their number of levels, they are referred as parallel plans. Searching optimal sequential plans is generally considered harder than searching optimal parallel plans. Büttner and Rintanen have proposed a search procedure which computes plans whose numbers of levels are fixed and whose numbers of actions are minimal. This procedure is notably used to calculate optimal sequential plans, starting from an optimal parallel plan. In this paper we describe a similar approach, which we have developed from the planner FDP*. The idea consists in maintaining two structures, the first one representing the parallel plan and the other representing the sequential plan, performing the choices simultaneously in both structures. The techniques which were developed in* FDP *to compute sequential plans or parallel plans enable failures detection in the two structures. Experimental results show that this approach is in some cases more efficient than* FDP *when searching optimal sequential plans.*

1. Introduction

The planner FDP [1] was presented for the first time at IPC5 planning competition in 2006. This planner has the particular ability to generate optimal sequential plans. It is generally admitted that searching such plans is more costly than searching optimal parallel plans. Indeed, parallel plans are shorter and then they are more *constrained*, which makes the search space smaller. As for many other optimal planners, the proof of the optimality comes from the proof that there is no plan with less actions or less steps. Then the method consists in searching a plan with one step, then a plan with two steps, and so on, until a solution is found or a given maximal length is reached. FDP uses its own search procedure, a depth first search progressing from the initial states towards the goals. This makes it simple and easy to adapt and to control the search. FDP employs an Iterative Deepening Depth First Search [2], that is successive depth-limited depth first searches, increasing the limit iteratively.

FDP achieves decent results at IPC5 in the category of optimal planners (deterministic track). Most planners in this category generate parallel plans. The efficiency of FDP is due to the various techniques which are used in its search procedure: in particular the order constraint helps to avoid the generation of redundant actions sequences (this technique is similar to *commutativity pruning* [3]), the unreachability of the goals is evaluated each time a new state is discovered, and the unsuccessfull searches are memorized in a hash table, so that these situations are not processed when they are encountered again.

There are few works on searching optimal sequential plans. In 2005, Büttner and Rintanen [4] have proposed a novel approach. The method consists, starting with an optimal parallel plan, in decrementing the number of actions while increasing the number of parallel steps, until the number of parallel steps is greater than the number of actions. Each time, the problem of determining if there exists a plan with at most n actions and m parallel steps is encoded as a SAT formula, and an external solver is used to search for a plan. The constraints on the number of actions and on the number of parallel steps make the search space smaller, and in many cases the SAT formula is satisfiable and easy to solve. In this paper, we propose a new procedure that searches for plans constrained on their numbers of actions and parallel steps, based on the works of S. Grandcolas and C. Pain-Barre [1] on the one hand, and G. Gabriel and S. Grandcolas [5] on the other hand. The idea consists in maintaining simultaneously two structures, one representing the sequential plan and the other the parallel plan. The techniques developped in FDP are applied in the two structures. During the search both structures may detect failures and cause backtrack. We have applied this procedure to the computation of optimal sequential plans, starting with an optimal parallel plan, and we have compared it with FDP.

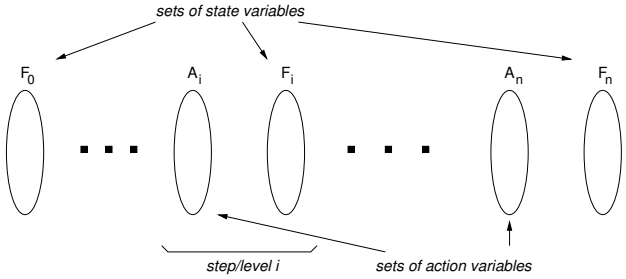
In the second part of this paper we present briefly FDP and its adaptation to the computation of parallel plans. The third part contains the description of the procedure which searches plans constrained on their number of actions and on their number of parallel steps. The fourth part is devoted to the search of optimal sequential plans starting with an optimal parallel plan as it was proposed by Büttner and Rintanen. Finally in the fifth part of the paper we give some experimental results.

2. FDP-structures

A planning problem is a triple $\mathcal{P} = (I, G, A)$, where I is the initial state, G is the set of the goals to achieve, and A is the set of the available actions. From $\mathcal{P}$ is derived the set F of the problem's fluents. An action a of A is given by its preconditions $pre(a)$ and its effects $\mathit{eff}(a)$. An action *adds* a fluent f if $f \in \mathit{eff}(a)$, and *removes* it if $\neg f \in \mathit{eff}(a)$.

Basically, the core of FDP is a decision function that answers the following question: does there exist a solution of length n for the problem $\mathcal{P}$? In sequential planning, n is the number of actions of the plan. In parallel planning, it is the number of its steps, where each step is a nonempty set of *compatible* actions. To answer this question, FDP makes use of a CSP-like structure of size n, called *fdp-structure*, that contains a set of potentially valid plans of length n. The search mainly consists in removing or fixing actions in this structure, until it solely contains a valid plan achieving G, or until a failure is proven.

The *fdp-structure* is composed of a set of variables that are subject to constraints. The variables are partitioned into sets of state variables and sets of action variables. A *fdp-structure* of size n contains $n+1$ sets of state variables and n sets of action variables. It can be seen as a leveled graph, similar to the one of Graphplan [6], where a step (or level) i contains a set of action variables A_i and a set of state variables F_i:



The set F_i represents the state after i steps: for each fluent $f \in F$, a state variable f_i indicates if it is *true*, *false* or *undefined*[1]. If it is undefined, the state is also partially (un)defined. If no variables of F_i are undefined, then F_i is actually a state. The initial state is represented by F_0 and is completely defined, according to the closed world assumption.

The set A_i encodes the actions of the step i. It differs depending on the planning is sequential or parallel. In sequential planning, only one single action is allowed per step: thus, A_i is only composed of a single action variable a_i whose domain is a subset of A. In parallel planning, several actions can be executed within one step. Thus, A_i contains as many action variables than actions of the problem. Each of them indicates whether the corresponding action is retained to occur (is fixed), is rejected (removed), or may possibly occur at that step. We then define two fdp-structures: $S_{seq}(n) = (\{F_0^{seq}, \ldots, F_n^{seq}\}, \{A_1^{seq}, \ldots, A_n^{seq}\})$ for sequential planning and $S_{par}(n) = (\{F_0^{par}, \ldots, F_n^{par}\}, \{A_1^{par}, \ldots, A_n^{par}\})$ for parallel planning.

1. We actually make use of two variables for a single fluent and a step: f_i for f and $\neg f_i$ for $\neg f$. For clarity, in the rest of the paper, we will not mention theses variables, unless necessary.

Before searching for a solution of length n, every state variables have $\{true, false\}$ as domains, i.e. the corresponding fluent is undefined. In sequential planning, the domain of every action variables is the set of all the actions. In parallel planning, their domains are $\{true, false\}$. Hence, in both cases each action is initially possible at any step. Then, the domains of the variables of F_0 are reduced in order to match F_0 to the initial state I. Similarly, the domains of the state variables representing the goals in F_n are reduced, ensuring that G is actually true in F_n.

The domain reduction of a state variable is similar to the removal of a literal: Removing $true$ for f_i means that f is removed from the step i, as it cannot be true at that step. For convenience, f_i will be said to be true (false) if $true$ ($false$) is the only remaining value in its domain.

These removals make some values inconsistent for other variables, by propagation. Indeed, the variables of a fdp-structure are subject to some implicit constraints that are inherent to planning. As an exemple, if a literal is removed from a step, then the actions that have this literal as a precondition can be removed from the next step. Also, the actions that add this literal can be removed from the previous step.

This propagation is similar to maintaining arc consistency [7]. It is performed in FDP by the function MakeConsistent, that is called whenever the domain of a variable is reduced: at a first time when the structure is initialized, and afterwards during the search whenever a choice is made (that leads to a variable domain reduction). The function updates the fdp-structure so as to make it consistent, by removing literals and actions which become inconsistent, together with literals and actions becoming inconsistent because of these removals. A removal may lead to an inconsistent fdp-structure. For instance, when a literal and its opposite are inconsistent (they are both true, or both false), or in sequential planning when there are no more possible actions in a step. In such cases, MakeConsistent fails and the search procedure has to backtrack over the choice that caused the failure.

MakeConsistent makes use of the following rules in deducing the inconsistency of a literal or an action. One rule is only applicable in the case of parallel planning, and is annotated with '(par)'. Another one is only applicable in the case of sequential planning, and is annotated with '(seq)'. For convenience and simplicity, we will denote with A_i the set of possible actions but not yet chosen at the step i, and X_i will denote the set of chosen (fixed) actions at step i. In sequential planning, either X_i is empty or A_i is and X_i is a singleton.

Let $f \in F$ be a fluent and f_i its associated state variable at step i. The value $true$ (resp. $false$) for f_i is inconsistent if one of the following conditions holds:

- $\neg f_i$ is true (resp. false)
- $i > 0$ and $\exists a \in X_i$ s.t. $\neg f \in \mathit{eff}(a)$ (resp. $f \in \mathit{eff}(a)$)
- $i < n$ and $\exists a \in X_{i+1}$ s.t. $\neg f \in \mathit{pre}(a)$ (resp. $f \in \mathit{pre}(a)$)
- $i > 0, X_i = \emptyset$ and $\forall a \in A_i,\ \neg f \in \mathit{eff}(a)$ (resp. $f \in \mathit{eff}(a)$)
- $i < n, X_{i+1} = \emptyset$ and $\forall a \in A_{i+1},\ \neg f \in \mathit{pre}(a)$ (resp. $f \in \mathit{pre}(a)$)
- $i > 0, f_{i-1}$ is false (resp. true) and $\forall a \in X_i \cup A_i,\ f \notin \mathit{eff}(a)$ (resp. $\neg f \notin \mathit{eff}(a)$)
- $i < n, f_{i+1}$ is false (resp. true) and $\forall a \in X_{i+1} \cup A_{i+1},\ \neg f \notin \mathit{eff}(a)$ (resp. $f \notin \mathit{eff}(a)$)
- $i > 0, \exists f'_i$ s.t. f'_i is true and mutex(f_i, f'_i, i) (resp. mutex($\neg f_i$, f'_i, i)) or f'_i is false and mutex(f, $\neg f'_i$, i) (resp. mutex($\neg f_i$, $\neg f'_i$, i)

An action a is inconsistent in a step $i > 0$ if one of the following conditions holds:

- $\exists f \in \mathit{pre}(a)$, s.t. f_{i-1} is false
- $\exists f \in \mathit{eff}(a)$, s.t. f_i is false
- (par) $\exists a' \in X_i$ s.t. mutex(a, a', i)
- (seq) $\exists f \in F$ s.t. f_{i-1} is false (resp. true), f_i is true (resp. false), and $f \notin \mathit{eff}(a)$ (resp. $\neg f \notin \mathit{eff}(a)$)
- $\exists f \in \mathit{pre}(a)$ s.t. $\neg f \notin \mathit{eff}(a)$ and f_i is false

Some rules are new in FDP: The last one, and those rules concerning the mutex relations. It should be noted that the mutex criterions for parallel planning are different from those of sequential planning. For parallel planning, we have adopted the Graphplan definition [6] although we extend it to negative literals: a positive literal could also be marked as mutex at a step with a negative literal, so could be two negative literals.

Since many years, sequential planning has received few attention. In particular, to the best of our knowledge, we do not know any work on mutex for sequential planning. Indeed, mutex have been mainly designed for parallel planning. Even the londex constraints [8] that extend mutex to capture exclusion between facts and actions across different time steps, are defined in the scope of parallel planning. In sequential planning, since only one action is allowed per step, all actions are pairwise mutex. This increases the number of mutex between literals. We then propose the following definition for the mutex relation between literals in the case of sequential planning.

In a sequential plan, two literals l_1 and l_2 are mutex at step i if one of the following conditions holds:

- l_1 and l_2 are two opposite literals
- l_1 and l_2 first appear at step i and no actions at this step add them together
- l_1 appears for the first time at step i whereas l_2 was previoulsy generated:
  - no actions of step i add them together, and
  - every action that adds l_1 at i :
    - either removes l_2,
    - or has l as a precondition but mutex(l, l_2, $i-1$) holds
- l_1 and l_2 have been previously generated: mutex(l_1, l_2, $i-1$) holds and no actions at i add them together, and every action at i that adds one of them removes also the other, or has a precondition that is mutex at step $i-1$ with the other

3. Searching a plan with m steps and n actions

The function Search determines if a plan with n actions and m steps exists, enumerating all possible plans simultaneously in the sequential structure S_{seq} and in the parallel structure S_{par}. This last one represents the sequential plan which is constructed, grouping together in each level the successive actions which are independent. In what follows, we shall use the term *step* in the context of sequential planning, and *level* in the context of parallel planning.

Each time a value is removed in one of these structures, this removal is also performed in the other structure. The search progresses from the initial state towards the final state. Then at any moment the states before the current step s in the sequential structure are all completely defined, and so are the states preceding the current level l in the parallel structure.

The function is initially called with the value 0 for s and l. In each call, the function tries to fix each possible action successively, that is actions which occurs in the sequential structure at the step s and in the parallel structure at the level l, and the function is called recursively to determine if the current choice leads to a valid plan. If no solution is discovered with these actions, then the actions which occur at the step s and not at the level l of the structure S_{par} but at the level $l+1$ are considered. In this case, since a new level is entered, the remaining possible actions at level l must be removed. Each time some actions are fixed or removed in a structure, the consequences of these changes are propagated through the structure, calling the function MakeConsistent. The choice of the action a fails if this choice is inconsistent in the sequential structure or in the parallel structure.

Failures memorizing. FDP memorizes the failures which are encountered during the search, in order to avoid similar useless searches. Each time the planner proves that it is impossible to reach the goals from the current state F with the remaining steps, the pair $\langle F, d\rangle$ where d is the number of remaining steps is memorized. This indicates that if the current state is equal to F and if there are no more than d remaining steps then the search may be abandoned.

We propose here to memorize failures in a similar way. Each time the procedure enters a new level in the parallel structure, the triple $\langle F, na, ns\rangle$ where F is the current state (it is the same state in the sequential structure), na is the number of remaining steps in the sequential structure and ns is the number of remaining steps in the parallel structure, is registered. This memorization is not very efficient: many choices are performed within the same level in the parallel structure. Then few states are memorized, and failures may be detected very lately. A solution consists in memorizing states each time a possible action is fixed. Unfortunately this memorization would make the procedure incomplete:

Function `Search`(S_{seq}, s, n, S_{par}, l, m)

Data: $S_{seq} = (A^q, F^q, n)$ a FDP-structure of length n, the first s steps are instanciated,
Data: $S_{par} = (A^p, F^p, m)$ a FDP-structure of length m, the first l steps are instanciated,
Result: $TRUE$ if there exists a plan in these structures, $FALSE$ in the other case.

```
begin
    if s > n or l > m then
        return TRUE;
    C := A_s^seq ∩ A_l^par;
    for a ∈ C do
        remove all actions from A_s^seq but a;
        if not MakeConsistent(S_seq) then
            goto REVERT_SEQ;
        if s + 1 ≤ n and ⟨F_{s+1}^seq, A_{s+1}^seq ∩ A_l^par, n − (s + 1), m − l⟩ ∈ H then
            goto REVERT_SEQ;
        fix the action a at step l in S_par;
        if MakeConsistent(S_par) then
            if Search(S_seq, s + 1, n, S_par, l, m) then
                return TRUE;
            if s + 1 ≤ n then
                H := ∪{⟨F_{s+1}^seq, A_{s+1}^seq ∩ A_l^par, n − (s + 1), m − l⟩};
        Revert(S_par);
      REVERT_SEQ:
        Revert(S_seq);
    if l < m then
        remove unfixed actions from A_l^par;
        remove C actions from A_s^seq;
        if MakeConsistent(S_par) and MakeConsistent(S_seq) then
            if Search(S_seq, s, n, S_par, l + 1, m) then
                return TRUE;
        Revert(S_seq), Revert(S_par);
    return FALSE;
end
```

suppose for example that fixing the action a in the parallel structure implies the removal of the action b at the same level, because b removes a precondition of a (figure 1). Suppose now that the state F resulting of the execution of a in the sequential structure does not lead to the goals and then must be memorized. Since the action b has been removed from the parallel structure, it can not be chosen in the sequential structure at step $s + 1$ although b preconditions are in F. If the goals are reachable from F using the action b this memorization may prevent the discovering of a solution.

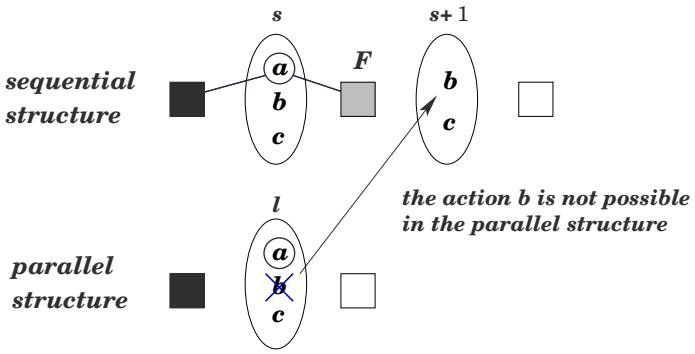


Fig. 1. memorizing F

To remedy this problem, we propose to memorize the state F together with the set U of the actions which are possible at the same time in the sequential structure and in the parallel structure (figure 2). The failure is then represented by the quadruplet $\langle F, U, na, ns\rangle$. In figure 2 the actions b and d at the level l have been removed in the parallel structure, since they were not compatible with the current choices within that level (the actions a and f). Then they are not memorized.

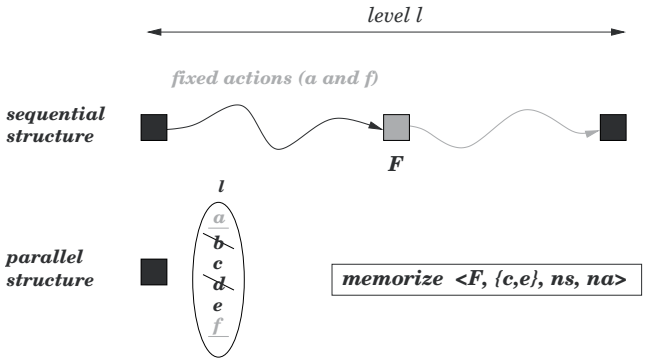


Fig. 2. memorizing F and the possible actions

The quadruplets which represent the failures are memorized in the table H. Each time an action is chosen in the sequential

structure, before trying to extend the current partial plan, the current state is searched in the table H. If the state occurs in the table with at least the actions which are possible at the same time in the sequential structure and in the parallel structure, and with greater numbers of remaining steps in the two structures, then the current search is abandoned since it can not lead to a valid plan.

4. Searching optimal sequential plans

The function SearchOptimalSequentialPlan follows the scheme proposed by Büttner and Rintanen [4]. Starting with an optimal parallel plan, the number of actions is decreased while a valid plan exists with these characteristics. If the number of steps is less than the number of actions the processus continues with one additional level for the parallel structure. In the other case the algorithm stops. The valid plan which was found in last is optimal in its number of actions.

Note that if the objective is only to search for an optimal sequential plan it is not necessary to explore all the optimal plans increasing one by one the number of levels in the parallel structure. One can simply perform a sort of dichotomic search on the number of levels, just to exhibit a valid plan containing the current number of actions. Indeed in many cases the algorithm performs many successive unsuccessfull searches, each time increasing the number of levels. The function DichotomicSearchOptimalSequentialPlan implements this strategy: the variable m_{min} represents the minimal number of levels for a plan of n actions. Then a plan is searched with $(m_{min} + n)/2$ levels. If it exists then the number of actions is decremented, and the minimal number of levels remains the same. In the other case this unsuccesfull search proves that $(m_{min} + n)/2$ levels are not enough for a plan with n actions, and then m_{min} is updated. With this method the function does not explore all the pairs (m, n). In particular, the optimal sequential plan which is produced by the algorithm is not always optimal in its number of levels: the last time the search is successfull, the number of levels $(m_{min} + n)/2$ is not necessarily minimal.

5. Experimental results

We have implemented the search for an optimal sequential plan (function SearchOptimalSequentialPlan) and we have compared it with FDP.

We did not compare our results with those of Büttner and Rintanen [4], because a comparison would be difficult. First, currently there is no available running version of their planner, and the only elements of comparison that we dispose of are those of their paper. However, they are very partial as they mainly concern some steps of some problems resolution. The total times of the problems resolution are only available for only four problems (Logistics8_0, Depot5, Mprime3 and Driverlog8), that are hardly handled by FDP. But a significant comparaison should require runs on many more series and problems.

Also, we have to note that their planner implements the $\exists$-step semantic for parallel plans, whereas PFDP implements the $\forall$-step semantic. The $\exists$-step semantic has been first proposed by Dimopoulos et al. [9] and has been defined and studied by Rintanen and al. in the case of propositional planning [10], and more recently in [11]. In a $\exists$-step plan, the actions of a time step are allowed to interfere, provided that they can be arranged such that their execution is possible. In contrast, the $\forall$-step semantic [10] is (a generalization of) the Graphplan semantic [6] : every ordering of the actions of a time step of a $\forall$-step plan must be possible and must lead to the same state. This reduces the set of solutions. As a consequence, when their planner find a plan for a problem given a number of steps and a number of actions, our planner might not. In particular, for most problems their starting optimal parallel plan is already shorter than ours. Although the optimal sequential plan is the same (if it is unique), we do not explore the same search space. This makes even harder the comparison with the available times in their paper.

Nevertheless, the $\exists$-step semantic is surely more suitable for the present approach, since it is very costly to prove insolvability of problems during the search. Our first aim was to implement the approach by making FDP and PFDP to cooperate. A first improvement would be to implement the $\exists$-step semantic.

The results are presented in figure 3. Times are in seconds on Linux computers with Pentium 3 GHz processor, a timeout of 3000 seconds, and 1 GB RAM. The problems come from the international planning competitions IPC-2, IPC-3, IPC-4 and IPC-5. For each problem the number of actions, the number of facts, the length of an optimal sequential plan and the number of levels of an optimal parallel plan are indicated.

The first version of the search procedure, denoted SPS for *search parallel sequential plans*, enumerates plans which are minimal in terms of the number of levels. Each time, the number of actions is fixed, starting with an initial value which is computed searching an optimal parallel plan, and which is decreased until there is more actions than levels (function SearchOptimalSequentialPlan). The second version that is proposed, denoted SPS-DICHO, implements a dichotomic search to determine for each length if a plan exists function DichotomicSearchOptimalSequentialPlan. Finally we have tested a simpler search procedure, denoted SPS-CTR, which consists in searching optimal sequential plans decreasing the number of actions of an initial optimal parallel plan, with no constraint on the number of levels. In this way we expect to make easier the first calls to the function Search, in which the number of levels is constraining, the number of actions is important and there are solutions. For information we have also reported the cpu times for the initial calculs of an optimal parallel plan, column PFDP, and the cpu times for the calculus of an optimal sequential plan with FDP.

Computation times include parsing of the problems, computing mutual exclusions and ordered sequences of actions, searching for an optimal parallel plan and searching for an optimal sequential plan.

Function `SearchOptimalSequentialPlan(`P`)`

Data: P a planning problem, S_{seq} and S_{seq} two FDP-structures,
Result: n_{opt} the length of an optimal sequential plan for the problem P.
begin
  suppose π is an optimal parallel plan with m steps and n actions for the problem P;
  $n_{opt} := n$;
  $n := n - 1$;
  while $m < n$ **do**
    initialize S_{seq} to search for a sequential plan of length n;
    initialize S_{par} to search for a parallel plan of length m;
    if *Search(*S_{seq}, 0, n, S_{par}, 0, m*)* **then**
      $n_{opt} := n$;
      $n := n - 1$;
    else
      $m := m + 1$;
  return n_{opt};
end

Function `DichotomicSearchOptimalSequentialPlan(`P`)`

Data: P a planning problem, S_{seq} and S_{seq} two FDP-structures,
Result: n_{opt} the length of an optimal sequential plan for the problem P.
begin
  suppose π is an optimal parallel plan with m steps and n actions for the problem P;
  $n_{opt} := n$;
  $n := n - 1$;
  $m_{min} := m$;
  while $m_{min} \leq n$ **do**
    $m = (m_{min} + n)/2$;
    initialize S_{seq} to search for a sequential plan of length n;
    initialize S_{par} to search for a parallel plan of length m;
    if *Search(*S_{seq}, 0, n, S_{par}, 0, m*)* **then**
      $n_{opt} := n$;
      $n := n - 1$;
    else
      $m_{min} := m + 1$;
  return n_{opt};
end

problem	**act.**	**facts**	**act.**	**niv.**	FDP	SPS	SPS-DICHO	SPS-CTR	PFDP
mprime-x-7	1728	426	5	5	11,4	19,43	19,37	19,36	19,16
mprime-x-9	1904	270	8	5	74,27	30,84	29,52	20,38	8,98
mprime-x-26	4594	287	6	5	61,52	61,34	61,33	61,4	58,27
mystery-x-2	3036	357	7	5	30,39	74,29	80,74	33,57	29,23
mystery-x-30	3357	408	9	6	87,48	113,49	109,87	79,18	71,06
Depot-7512	162	78	15	8	0,83	6,45	3,94	1,61	0,36
driverlog-2-2-3	108	57	19	9	8,23	25,87	15,04	6,06	0,08
driverlog-3-2-4	144	63	16	7	9,3	117	52,21	30,06	0,11
FreeCell3-4	1143	139	14	8	56,13	232,13	259,63	95,89	3,36
FreeCell4-4	1614	183	18	7	462,58	1813,01	2059,08	1260,91	5,79
FreeCell5-4	52	20	-	13	–	–	–	–	17,33
satellite-x-1	259	71	17	10	23,86	–	1043,52	343,86	249,33
Optical-P01-OPT2	418	282	36	13	49,57	156,89	60,23	13,05	5,79
Philosophers-P03-PHIL4	112	120	44	11	81,14	233,84	111,35	11,82	0,68
PSR-33	162	41	25	15	6,67	52,97	46,6	40,31	38,25
PSR-37	112	56	33	25	55,45	183,74	158,35	159,23	124,54
PSR-49	660	63	19	16	23,54	145,51	144,89	158,78	144,48
pipesworld-n1-14-6	632	139	13	8	77,36	496,96	438,75	175,43	20,48
pipesworld-n2-10-2	720	201	20	12	140,3	294	157,71	71,67	11,19
pipesworld-n3-12-2	1140	280	14	14	13,58	1475,4	1477,46	1478,81	1483,32
pipesworld-p04	656	154	11	6	41,73	67,14	49,47	20,76	3,61
pipesworld-p06	764	164	10	6	6,43	16,74	15,02	8,61	4,8
pipesworld-p07	2672	204	8	6	33,73	297,7	294,78	295,1	180,03
Storage-11	460	146	17	11	41,47	173,11	93,67	51,22	16,13
Storage-12	690	164	16	9	214,41	1705,17	877,45	329,35	45,87
Truck-7	1044	269	23	18	714,18	1600,07	1266	671,44	394,37

Fig. 3. CPU times for a selection of problems (times are in seconds)

First remark that in most cases, with FDP like approaches, the computation of an optimal parallel plan is faster than the computation of an optimal sequential plan (see also [5]). This is due to the fact that parallel plans are shorter, and then the search space is smaller: states and actions levels are closer to the goals and then the instanciation of the goals is more effective. Cases in which FDP is more efficient often correspond to problems in which optimal parallel plans are quite sequential (there are very few actions at each level). Remind also that FDP makes use of a very efficient heuristic to evaluate if the goals are not reachable with the remaining actions, and consistency rules are stronger in FDP.

Searching an optimal sequential plan, starting with a parallel one, is rarely attractive. If the aim is to go fast, then the SPS-DICHO and SPS-CTR methods are the best. It is natural to think that the cost of successive searches which stop with a success, which is the case of the SPS-CTR method, is less than the cost of successive unsuccessfull searches, which is the case with FDP, since the first solution which is encountered is an optimal plan.

In some cases there is a combinatorial explosion for the computation or the proof of optimality of a sequential plan, as it is the case for Free-Cell or Storage problems. For other problems, like PSR and some PipesWorld problems, it is the opposite. One could imagine that it would be usefull to call simultaneously many search procedures following various strategies, so as to minimize the risks.

Finally we focused on a particular domain, the airport domain, which consists in controling the ground traffic on airports. This domain has been developed by Jörg Hoffmann and Sebastian Trüg for the IPC-4 competition. The figure 4 shows the results for the twenty easiest problems of the serie. It is clear that SPS-DICHO performs better than FDP. Moreover harder are the instances more important is the gain. A particularity of the problems in which the gain is important is that the optimal sequential plans are much longer than the optimal parallel plans. On the other hand the optimal parallel plans are often also optimal sequential plans. In these cases the procedure has just to verify that there is no plan with one action missing, whatever is the number of steps. The procedure SPS-CTR and in a lesser measure the procedure SPS-DICHO have not to perform many unsuccessfull searches as has the procedure SPS.

Finally remark that two problems of the serie were solved by SPS-CTR within the timeout of 3000 seconds while FDP did not find a solution.

6. Conclusion

We have presented a procedure based on FDP search procedure, which searches for plans in which the number of actions and the number of steps are constrained. Two structures are used simultaneously, the first one represents the sequential plan and while the other represents the parallel plan. We have experimented the application of this procedure to search optimal sequential plans, starting with an optimal parallel plan, following the idea of Büttner and Rintanen [4]. This way, the search explores the pairs (m, n) where n is the minimal number of actions for a parallel plan of length m. In particular the search yields an optimal sequential plan. We have implemented our approach and we have compared it with the optimal sequential planner FDP for the computation of an optimal sequential plan. Our results are in general worse than the results obtained with FDP, but they remain competitive.

Finally we have implemented a version of the search in which the length of the parallel plan is not constrained, the initial optimal parallel plan providing the initial number of actions, which is very efficient. In further works, we will explore this strategy, in which case most searches are successfull, in opposition with FDP which starts searching short plans. Furthermore the implementation of $\exists$-step semantic for parallel plans should be of great help in that direction. We will also study how to adapt our search procedure to optimize a function depending on the number of actions and on the makespan.

References

[1] S. Grandcolas and C. Pain-Barre, "Filtering, decomposition and search space reduction for optimal sequential planning," in *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*, july 2007, pp. 993–998.

[2] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, pp. 97–109, 1985.

[3] P. Haslum and H. Geffner, "Admissible heuristics for optimal planning." in *AIPS*, 2000, pp. 140–149.

[4] M. Büttner and J. Rintanen, "Improving parallel planning with constraints on the number of operators," in *Proceedings of the 15th International Conference on Automated Planning and Scheduling*, 2005, pp. 292–299.

[5] G. Gabriel and S. Grandcolas, "Searching optimal parallel plans: A filtering and decomposition approach," in *proceedings of the 21st International Conference on Tools with Artificial Intelligence*, 2009, pp. 576–580.

[6] A. Blum and M. Furst, "Fast planning through planning graph analysis," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, 1995, pp. 1636–1642.

[7] A. Mackworth, "Consistency in networks of relations," in *Artificial Intelligence*, 1977, pp. 8:99–118.

[8] Y. Chen, Z. Xing, and W. Zhang, "Long-distance mutual exclusion for propositional planning," in *IJCAI'07: Proceedings of the 20th international joint conference on Artifical intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, pp. 1840–1845.

[9] Y. Dimopoulos, B. Nebel, and J. Koehler, "Encoding planning problems in nonmonotonic logic programs," in *ECP '97: Proceedings of the 4th European Conference on Planning*. London, UK: Springer-Verlag, 1997, pp. 169–181.

[10] J. Rintanen, K. Heljanko, and I. Niemelä, "Planning as satisfiability: parallel plans and algorithms for plan search," *Artif. Intell.*, vol. 170, no. 12-13, pp. 1031–1080, 2006.

[11] M. Wehrle and J. Rintanen, "Planning as satisfiability with relaxed ∃-step plans," in *AI'07: Proceedings of the 20th Australian joint conference on Advances in artificial intelligence*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 244–253.

problem	act.	facts	act.	niv.	FDP	SPS	SPS-DICHO	SPS-CTR	PFDP
Airport-1	15	80	8	8	0,08	0,09	0,09	0,09	0
Airport-2	23	81	9	9	0,08	0,11	0,1	0,1	0
Airport-3	38	131	17	9	0,33	0,41	0,38	0,37	0,3
Airport-4	23	156	-	500	0,31	0,23	0,23	0,22	0,2
Airport-5	54	197	21	21	2,85	2,87	2,88	2,86	2,8
Airport-6	77	291	41	21	6,68	7,9	6,99	6,68	6,4
Airport-7	77	291	41	21	6,68	7,87	6,97	6,67	6,4
Airport-8	131	412	62	26	74,78	117,94	45,68	22,2	13,3
Airport-9	143	483	71	27	578,53	1367,91	281,56	93,36	18,3
Airport-10	29	178	18	18	0,47	0,49	0,5	0,48	0,4
Airport-11	60	219	21	21	3,55	3,58	3,57	3,54	3,5
Airport-12	89	327	39	21	9,62	10,62	9,76	9,46	9,3
Airport-13	87	322	37	19	7,98	8,87	8,22	7,94	7,7
Airport-14	149	462	60	26	81,77	134,63	46,7	30,04	18,6
Airport-15	147	459	58	22	73,53	130,31	44,87	28,73	17,3
Airport-16	207	594	79	27	2990,31		1090,05	408,66	35,2
Airport-17	225	687	88	28				2795,95	77,1
Airport-18	283	811	107	31					888,7
Airport-19	229	755	90	30				1906,49	103,1
Airport-20	302	898	115	32					3219,4

Fig. 4. CPU times for a selection of Airport problems (times are in seconds)