Filtering, Decomposition and Search Space Reduction for Optimal Sequential Planning

Stéphane Grandcolas, Cyril Pain-Barre

Filtering, Decomposition and Search Space Reduction for Optimal Sequential Planning

Stéphane Grandcolas and **Cyril Pain-Barre**
LSIS – UMR CNRS 6168
Domaine Universitaire de Saint-Jérome
Avenue Escadrille Normandie-Niemen
F-13397 Marseille Cedex 20 - France
stephane.grandcolas@lsis.org, cyril.pain-barre@lsis.org

Abstract

We present in this paper a hybrid planning system which combines constraint satisfaction techniques and planning heuristics to produce optimal sequential plans. It integrates its own consistency rules and filtering and decomposition mechanisms suitable for planning. Given a fixed bound on the plan length, our planner works directly on a structure related to Graphplan's planning graph. This structure is incrementally built: Each time it is extended, a sequential plan is searched. Different search strategies may be employed. Currently, it is a forward chaining search based on problem decomposition with action sets partitioning. Various techniques are used to reduce the search space, such as memorizing nogood states or estimating goals reachability. In addition, the planner implements two different techniques to avoid enumerating some equivalent action sequences. Empirical evaluation shows that our system is very competitive on many problems, especially compared to other optimal sequential planners.

Introduction

In the domain of planning, many approaches first convert the problem into other formalisms of Artificial Intelligence. Generally, a bound on the length of the searched plan is specified, so that the problem is in NP. The first systems based on these ideas used SAT encodings of the problems, and required SAT solvers. The relative success of these approaches has led researchers to work on even better SAT encodings (Kautz & Selman 1999), and to explore other formalisms such as Integer Programming (Vossen *et al.* 1999) or Constraint Satisfaction (van Beek & Chen 1999).

We propose in this paper a new constraint satisfaction approach for planning. Usually planning as constraint satisfaction consists in translating a planning problem into a CSP, and then in using a CSP solver to search for a solution. The performances of the system depends on the CSP encoding and on the efficiency of the solver. The encoding may be either hand-made, as in CPLAN (van Beek & Chen 1999), or automatically generated, often starting from the well-known planning graph of Graphplan (Blum & Furst 1995), as in GP-CSP (Do & Kambhampati 2001). Most systems generate *mutual exclusion constraints*, which help to reduce the search space, the way BLACKBOX (Kautz & Selman 1999) or SATPLAN did with SAT encodings. Recent systems generate even more constraints, deriving the CSP encoding directly from the problem definition (Lopez & Bacchus 2003). Although these approaches enable the use of up-to-date solvers, heuristics rules and pruning properties which are specific to the domain are not easily coded.

In fact, some systems handle SAT/CSP encodings but do not use external solvers, thus controlling completely the search. For instance, Rintanen planning algorithm (Rintanen 1998) performs undirectional search on a SAT encoding. The system generates new clauses which represent *invariants*, and which help to better propagate assignments of the propositional variables. DPPLAN (Baioletti, Marcugini, & Milani 2000) works also on a SAT encoding of the planning graph, implementing a specific Davis and Putnam search procedure with different strategies.

The planning system which is described in this paper search for optimal sequential plans. It combines constraint satisfaction techniques like arc-consistency, with specific planning techniques, that help to prune the search space. The planner works on a structure similar to a planning graph. This structure is incrementally extended until a solution is found or a fixed bound on the length of the plan is reached. To improve the search, state-space planning heuristics may be employed, e.g. (Haslum, Bonet, & Geffner 2005; Hoffmann & Nebel 2001). Furthermore the consistency rules and the filtering procedure are compatible with any decomposition mechanism of the structure. Hence, it is possible to perform undirectional search just as well any other approach to planning as constraint satisfaction or satisfiability do, e.g. (Rintanen 1998; Baioletti, Marcugini, & Milani 2000; Lopez & Bacchus 2003).

Our search procedure is complete in the sense that the solutions are optimal plans (that is they are minimal in the number of actions since our planner generates only sequential plans). This feature is quite rare for nowadays planners, e.g. (Hoffmann & Nebel 2001; Blum & Furst 1995; Do & Kambhampati 2001), but may be of interest for some applications, in particular when the actions have costs. A future development of this work will take into account valued actions, so as to produce plans with minimal costs.

The paper is organized as follows: first we define the planning structure on which the system works, then we introduce the consistency rules and we present the filtering procedure,

next the search procedure and some techniques to reduce the search space are described. Finally, we present some experimental results that show the competitiveness of our planner compared to other optimal sequential planners, and we conclude with some future developments.

Framework and problem representation

The purpose of planning is to find a *plan*, which is a set of actions taken in a given set A and an (partial or total) ordering of them, such that the execution in a given *initial state* I of any sequence of the plan's actions according to the ordering achieves given *goals* G.

Definition 1 (Planning Problem) *A planning problem $\mathcal{P}$ is a 4-tuple (A, I, G, L) where A is a set of actions, I is the initial state, G are the goals, and L is the set of all the literals constructed from any proposition that occur in I, G and in A (in positive or negative form).*

I and G are conjunctions of literals and I is interpreted according to the *closed world assumption*. The actions are grounded (but the input language is PDDL with typing and equality) : Every action $a \in A$ is described by its preconditions $\mathsf{pre}(a)$, and its effects $\mathsf{eff}(a)$. $\mathsf{pre}(a)$ is a set of literals that must be true for a to be applicable, and $\mathsf{eff}(a)$ is a set of literals that a makes true.

Nowadays, many planners return a solution plan as a sequence of actions sets, where the actions of the same set can be executed in any order, even in parallel. The planner we present produces plans as totally ordered sequences of actions that are optimal in terms of the number of actions, what is usually not guaranteed by parallel approaches. The system makes use of special CSP-like representations to find valid plans of a given length, called *planning-structures*.

Definition 2 (planning-structure) *Given a planning problem $\mathcal{P} = (A, I, G, L)$, let define a planning-structure for $\mathcal{P}$ as a 4-tuple $\langle k, V_a, V_l, d\rangle$, where:*

- *k is the size of the planning-structure,*
- *$V_a = \{y_0, \ldots, y_{k-1}\}$ is a set of* action variables,
- *$V_l = \{x_{i,l}\}_{0 \leq i \leq k, l \in L}$ is a set of* literal variables,
- *d is a function returning the variables domains :*
  - *$\forall y_i \in V_a$, $d(y_i) \subseteq A$, is denoted A_i,*
  - *$\forall x_{i,l} \in V_l$, $d(x_{i,l}) \subseteq \{\top, \bot\}$, is denoted $D_{i,l}$.*

A planning-substructure *of $\langle k, V_a, V_l, d\rangle$ is a 4-tuple $\langle k, V_a, V_l, d'\rangle$ such that for each $x \in V_a \cup V_l$, $d'(x) \subseteq d(x)$.*

Any literal variable whose domain is $\{\top, \bot\}$ is *undefined*. If v is the value $\top$ (resp. $\bot$), its *opposite* – denoted $opposite(v)$ – is the value $\bot$ (resp. $\top$). A planning-structure can be viewed as a graph similar to a planning graph (Blum & Furst 1995). It is a leveled graph that alternates *literal levels* and *actions levels*. The i-th literal level, denoted F_i, represents the validity of all the literals at step i: it is the set of the literal variables $\{x_{i,l}\}_{l \in L}$. It is important to note that there is a close relationship between any literal variable in a level and the variable representing its opposite in the same level. Indeed, for any $x_{i,l}$ with domain $D_{i,l}$, the domain $D_{i,\neg l}$ of $x_{i,\neg l}$ should be $\{opposite(v) \mid v \in D_{i,l}\}$ (see definitions 3.3 and 4.1). A **state** is then defined as any literal level whose literal variables are assigned a value. The i-th actions level A_i represents the possible values for the action that is to be applied at step i. Note that a planning-structure does not contain *no-op* actions.

Definition 3 (Valid Plan) *A* valid plan *for the planning-structure $\langle k, V_a, V_l, d\rangle$ is an assignment θ of the variables in $V_a \cup V_l$ such that:*

1. $\forall y_i \in V_a,\ \theta(y_i) \in A_i$,
2. $\forall x_{i,l} \in V_l,\ \theta(x_{i,l}) \in D_{i,l}$,
3. $\forall x_{i,l} \in V_l,\ \theta(x_{i,\neg l}) = opposite(\theta(x_{i,l}))$,
4. $\forall l \in I,\ \theta(x_{0,l}) = \top$, *and* $\forall p \notin I,\ \theta(x_{0,p}) = \bot$, *where p is a proposition*
5. $\forall l \in G,\ \theta(x_{k,l}) = \top$,
6. $\forall y_i \in V_a, \forall l \in \mathsf{pre}(\theta(y_i)),\ \theta(x_{i,l}) = \top$,
7. $\forall y_i \in V_a, \forall l \in \mathsf{eff}(\theta(y_i)),\ \theta(x_{i+1,l}) = \top$,
8. $\forall l \in L$, *if* $\theta(x_{i,l}) \neq \theta(x_{i+1,l})$ *then* $l \in \mathsf{eff}(\theta(y_i))$ *or* $\neg l \in \mathsf{eff}(\theta(y_i))$.

Searching for an optimal sequential plan consists in searching for a valid plan with the smallest k.

Consistent planning-structures

Given a planning-structure, consistency rules aim to remove values in the variables domains that cannot occur in any valid plan. For example an action whose one precondition can't be true should not be considered, and then can be removed without loss of completeness.

These built-in rules capture all the axioms of satisfiability approaches to sequential planning. They correspond roughly to the propagation rules of DPPLAN, with the major difference that DPPLAN is a parallel planner (which does not have the same heuristics for reducing the search space).

Definition 4 (Inconsistent values of literal variables) *Given the planning-structure $\langle k, V_a, V_l, d\rangle$, the value $\top$ for the literal variable $x_{i,l} \in V_l$ is inconsistent if any of the following situations holds:*

1. **(logical consistency)** $0 < i \leq k$, $\bot \notin D_{i,\neg l}$
2. **(forward persistence)** $0 < i \leq k$, $\top \notin D_{i-1,l}$ *and* $\forall a \in A_{i-1},\ l \notin \mathsf{eff}(a)$,
3. **(all actions delete)** $0 < i \leq k,\ \forall a \in A_{i-1},\ \neg l \in \mathsf{eff}(a)$,
4. **(backward persistence)** $0 \leq i < k$, $\top \notin D_{i+1,l}$ *and* $\forall a \in A_i,\ \neg l \notin \mathsf{eff}(a)$,
5. **(opposite always required)** $0 \leq i < k, \forall a \in A_i,\ \neg l \in \mathsf{pre}(a)$

One can note that the fifth rule does not exist in DPPLAN.

Definition 5 (Inconsistent values of action variables) *The value $a \in A_i$ for the action variable $y_i \in V_a$ is inconsistent if one of the following situations holds:*

1. **(falsified precondition)** $\exists l \in \mathsf{pre}(a)$ *such that* $\top \notin D_{i,l}$,
2. **(falsified effects)** $\exists l \in \mathsf{eff}(a)$ *such that* $\top \notin D_{i+1,l}$,

3. **(effect required)**
$\exists l \in L$ *such that* $\top \notin D_{i,l}$, $\bot \notin D_{i+1,l}$ *and* $l \notin \mathsf{eff}(a)$.

In the following we will distinguish values for literal variables with those of action variables only when necessary. A value that is not inconsistent is consistent. One can note that the third rule does not exist in DPPLAN since it is a parallel planner.

Definition 6 (Consistent planning-structure) *A planning-structure* $\langle k, V_a, V_l, d\rangle$ *is* consistent *if and only if:*

1. *no domain values are inconsistent*
2. *no domains are empty*

Property 1 (Consistency in a valid plan) *Given a valid plan* θ *for a planning-structure* $\langle k, V_a, V_l, d\rangle$*, then for each* $x \in V_a \cup V_l$ *the value* $\theta(x)$ *is consistent.*

Property 2 (Largest consistent substructure) *A planning-structure* S *is equivalent (i.e. represents the same valid plans) to its largest consistent substructure* S' *if one exists, and this substructure is unique. If no consistent substructure exists then there is no valid plans for* S.

Formalizing planning-structures as constraint satisfaction problems is feasible, for example making use of n-ary constraints or dynamic CSP (van Beek & Chen 1999; Do & Kambhampati 2001). Since our objective here is just to reduce the structure removing inconsistent values, we make the choice to use specific representation and filtering procedures.

Filtering inconsistent values and actions

Making a planning-structure consistent consists in removing inconsistent values until none exists or a domain becomes empty. The function **MakeConsistent** (not detailed in this paper) makes the necessary removals in a given planning-structure and returns either TRUE or FALSE whether the planning-structure is consistent or not. Similarly to arc consistency enforcing procedures in the domain of constraint satisfaction (Mackworth 1977), the *"values to remove"* elements are stored in a queue H. Until H is empty, an element is extracted, the corresponding value is removed, and the values which become inconsistent because of this deletion are enqueued in H. The inconsistency rules are used to detect these new inconsistencies (cf. definitions 4 and 5). Maintaining counters and lists for actions preconditions and effects and for literals allows to efficiently detect new inconsistencies. The removals are then propagated forward and backward through the planning-structure.

If a domain becomes empty, the function stops and returns FALSE. In the other case the function stops with a consistent planning-structure S.

Search procedure and search space reduction

We now present the current search procedure. The system starts searching from a planning-structure of length 1 and increments its length until a plan is found or a given fixed bound is reached. One can note that an improvement of the procedure would be to start searching from an initial length larger than 1, and to extend the structure with more than one step after a failure. This can be achieved for example by using the (additive) h^m family of heuristics (Haslum & Geffner 2000; Haslum, Bonet, & Geffner 2005). This will be addressed in a future work.

When searching for a plan of length k, a planning-structure S of length k is constructed. Initially each action set of S is set to A, and each literal variable is undefined. Then, the values which are not in the initial state and the opposites of the goals are removed and a preliminary filtering is performed on S. If S is inconsistent then the search stops with failure, there is no plan of length k. In the other case the function **Search-Plan** is called with the planning-structure S. This function returns TRUE if the planning structure contains a valid plan, and returns FALSE otherwise.

A *divide and conquer* approach is employed to search for a plan in the planning-structure: The structure is decomposed into smaller substructures and the procedure searches recursively each of them. The substructures are systematically filtered, in order to detect failures as soon as possible. Several decomposition procedures have been experimented, such as *enumerating actions*, *assigning literal variables* or *splitting action sets*. The latter gives the best results. Splitting consists in partitioning a set of actions so as to put together actions which have deletions in common. The splitting procedure search for an undefined literal variable such that the number of actions that delete it and the number of actions that do not are as close as possible. The structure S is decomposed into two substructures corresponding to the partition of an action set (line 12), and each substructure is searched recursively (line 13 and line 15).

In fact, **Search-Plan** is a depth first iterative deepening search, since it always chooses the first non singleton action set for splitting, starting from the initial state (loop line 3 to line 11). This ensures the optimality of the solution plan if one exists. Then the current partial plan is extended step by step until the planning-structure becomes inconsistent or a valid plan is found (line 4: Since S is consistent, if all the action sets are singletons then no literal variables are undefined in S, and S is a plan).

We now explain some techniques for pruning the search space.

Nogood recording

During the search, each time a state is encountered and the current call returns failure, this state is memorized as a nogood in a hash-table. The distance $k - i$ from F_{i+1} to the final state is also memorized. Indeed, it is always possible that the state F_{i+1} could be extended to a valid plan if there were more steps. Whenever F_{i+1} will be encountered at the distance less than or equal to $k - i$ to the final state the search will be aborted (line 6). Nogoods are preserved each time the structure is extended. In addition, the memorization is done before the result of the current call is established (line 8). Memorizing nogoods improves drastically the performances of the search as it does in many other planners (Blum & Furst 1995; Kambhampati 2000). A possible improvement would be to restrict the memorization to the literals which are involved in the failures.

function **Search-Plan**(i, S, k)
$\{S$ *is a* k*-steps structure*$\}$
$\{i$ *is the current step (*$A_0, \ldots, A_{i-1}$ *are singletons)*$\}$

1 **if not** MakeConsistent(S) **then**
2 **return** FALSE.
3 **while** $|A_i| = 1$ **do**
4 **if** $i = k - 1$ **then**
5 **return** TRUE, (S *is a valid plan*)
6 **if** $\langle F_{i+1}, k - i\rangle \in$ NoGoods **then**
7 **return** FALSE,
8 NoGoods := NoGoods $\cup\{\langle F_{i+1}, k - i\rangle\}$,
9 **if** GoalsAreUnreachable$(S, i + 1, k)$ **then**
10 **return** FALSE,
11 $i := i + 1$,
12 $\langle A', A''\rangle :=$ SplitActionSet(A_i),
13 **if** Search-Plan$(i, S_{A_i \leftarrow A'}, k)$ **then**
14 **return** TRUE,
15 **if** Search-Plan$(i, S_{A_i \leftarrow A''}, k)$ **then**
16 **return** TRUE,
17 **return** FALSE.

Estimating goals reachability

Anytime a literal level F_i is a state, the system checks if the goals are possibly achievable. It is done by selecting at each of the remaining steps the action which adds the most goals which are not in F_i. If the total number of goals added by the selected actions is less than the number of missing goals at step i, then it is not possible to achieve all the goals and the search procedure backtracks (line 9). **GoalsAreUnreachable** returns TRUE if it can prove that achieving all the goals from the current state is not possible.

Mutually exclusive literals and actions

Most planning systems take benefit from mutual exclusions constraints between literals and between actions. Our current implementation does not use such constraints. In sequential planning, the actions are mutually exclusive. Mutual exclusion of literals may receive a particular attention. Indeed, if two literals l and l' are marked as being mutually exclusive in the sense of GRAPHPLAN, then the current filtering process removes value $\top$ of l whenever l' is assigned the value $\top$. However, they may be of interest in our scope as they can be used to remove actions that have mutually exclusive preconditions. This will be addressed in the future.

Equivalent action sequences

Since the system constructs sequential plans, it can enumerate equivalent permutations of actions (that is sequences of actions which produce the same state, starting from a given state) and perform as many redundant searches. To cope with this problem, we experiment two different techniques. With the first one, called **2-sequence ordering**, sequences which contain two successive actions which are "independent" and that do not respect an arbitrary order are discarded. With the second one, called **level ordering**, plans are viewed as series of sequences of independent actions. Each action is constrained to have a "predecessor", that is an action with which it is not independent, within the previous sequence. We first introduce the notion of admissible 2-sequence.

Definition 7 (Admissible 2-sequence) *The sequence of actions* (a, a') *is admissible if and only if one of the following situations holds:*

1. $\exists l \in \mathsf{pre}(a)$ *such that* $\neg l \in \mathsf{eff}(a')$,
2. $\exists l \in \mathsf{eff}(a)$ *such that* $\neg l \in \mathsf{eff}(a')$,
3. $\exists l \in \mathsf{eff}(a)$ *such that* $l \in \mathsf{pre}(a')$,

In the situation 1 the sequence (a', a) is not valid, and then the sequence (a, a') must be considered. In the situation 2, a and a' have some opposite effects, and then the sequences (a, a') and (a', a) are not equivalent. In situation 3 the sequence (a', a) may be inapplicable and for completeness the sequence (a, a') must be considered.

Admissible 2-sequences represent a sort of dependence between the actions. In particular an admissible 2-sequence is not equivalent to the inverse sequence. This notion is close to the relation of interference for mutual exclusive actions (see (Blum & Furst 1995) for example). Interference is not sufficient for our purpose since we can't discard sequences in situation 3. It is also related to *commutativity pruning* (Haslum & Geffner 2000).

Definition 8 (Ordered 2-sequence) *The sequence* (a, a') *is an ordered 2-sequence if and only if* a *and* a' *are not the opposite one of the other, and either the sequence* (a, a') *is admissible or both* (a, a') *and* (a', a) *are not admissible and* $a \prec a'$.

The 2-sequence ordering constraint is applied each time an action set A_i is reduced to a singleton. The actions at steps $i-1$ and $i+1$ which do not form valid sequences with the action at step i are removed. This filtering is not costly: the status of each pair of actions (either ordered 2-sequence or not) is computed once for all before the search starts and memorized in a table.

However, 2-sequence ordering is not efficient with equivalent permutations of more than two actions: consider for example three actions a_1, a_2, and a_3 such that $a_1 \prec a_2 \prec a_3$, and such that (a_3, a_1) is admissible, but none of the sequences (a_1, a_2), (a_2, a_1), (a_2, a_3) and (a_3, a_2) are admissible (in other words a_1 and a_2 are independent, and so are a_2 and a_3). The sequences $s_1 = (a_3, a_1, a_2)$ and $s_2 = (a_2, a_3, a_1)$ are equivalent, and both satisfy the 2-sequence ordering constraint. To tackle this problem we propose another approach called level ordering. First we define the predecessors and the level of an action.

Definition 9 (Predecessors and level of an action) *The action* a *is a* predecessor *of the action* a' *if either* (a, a') *is admissible, or*

4. $\exists l \in \mathsf{pre}(a)$ *such that* $\neg l \in \mathsf{pre}(a')$,

PRED(a) *denotes the set of the predecessors of the action* a.

The level *of the action* a_i *in the sequence* $s = (a_1, \ldots, a_n)$*, denoted* $level_s(a_i)$*, is recursively defined as follows: if* $E = \{a_1, \ldots, a_{i-1}\} \cap PRED(a_i)$ *is empty then* $level_s(a_i) = 1$ *else*

$$level_s(a_i) = 1 + \max_{a \in E} level_s(a)$$

The precedence relation extends the notion of admissible 2-sequence. In the previous example the levels of the actions of the sequences s_1 and s_2 are respectively $(1, 2, 1)$ and $(1, 1, 2)$. Note that moving backward an action in a plan until one of its predecessors is encountered gives a plan which is equivalent.

Definition 10 (Level-ordered sequences) *The sequence $(a_1, \ldots, a_n)$ is* level-ordered *if any two consecutive actions a_i and a_{i+1} verify $level(a_i) \leq level(a_{i+1})$ and, in the case $level(a_i) = level(a_{i+1})$, then $a_i \prec a_{i+1}$.*

A level-ordered sequence can be splitted into k consecutive subsequences $s_1, s_2, \ldots, s_k$, such that each sequence s_i contains only actions whose level is i. Any action in a subsequence s_i has a predecessor in the sequence s_{i-1}. As the actions within each subsequence are $\prec$-ordered, all the permutations but one of the subsequence are discarded.

The level ordering constraint is applied each time a new partial plan is constructed. If for instance, the planning-structure begins with a i-step plan (i.e. the action sets $A_0, \ldots, A_{i-1}$ are singletons), then the actions in A_i whose levels are less than the level of the action at step $i - 1$ are removed.

Relevant literals and actions

Searching for optimal plans, the actions which do not help effectively to achieve the goals are useless and should not be considered. In particular, this is the case of actions of the last step not adding goals. This property propagates backwards through the planning-structure.

Definition 11 (Relevant literals and actions) *In a given planning-structure $\langle k, V_a, V_l, d\rangle$,* relevant actions *and* relevant literals *are recursively defined as follows:*

1. *each goal in G is relevant at step k,*
2. *a literal l is relevant at step i if either it is relevant at step $i + 1$ or there exists an action $a \in A_i$ such that l is a precondition of a and a is relevant at step i,*
3. *an action a is relevant at step i if one of its effects is relevant at step $i + 1$.*

Actions that are not relevant at a given step are removed from this step as it could not serve in a minimal solution. It is quite a standard technique in AI planning. However, the relevance of the literals and actions is maintained during the search when removals are performed. Actions which loose their relevance are then removed.

Experimental results

We have compared our planner with some other planners on classical benchmark problems and International Planning Competition problems (IPC3, IPC4 and IPC5). We present here a very limited selection of problems (see `http://www.lsis.org/fdp/` for the full list of results). The objective is to exhibit the behavior of our planner according to the characteristics of the problems and to evaluate the interest of equivalent sequences discarding. For each instance the numbers of actions and facts of the planning-structure are listed (Figure 1). We also report the lengths of the optimal sequential plans and the makespans, that is the lengths of the optimal parallel plans. The makespans were obtained with SATPLAN (Kautz & Selman 1999) that we run with the Siege V4 SAT solver. All planners were run on the same machine with a timeout of 10,000 seconds.

Three versions of our planner have been tested: with 2-sequence ordering constraints, with level ordering constraints, and with no ordering constraints. For each version the total numbers of calls of the function **Search-Plan** and the total computation times are listed. We have also reported the results of BFHSP (Zhou & Hansen 2004) with a backward search (h3max option) and BFHSP with a forward search (h1max option), since BFHSP is at the moment one of the most impressive sequential planner. Instances which have resulted in some errors or timeout exceeding are reported with –.

In all cases the ordering constraints help to prune the search space, except for the xy-world problem. This problem is rather particular since its solutions involve only non-interfering actions. Then 2-sequence ordering discards all but one permutation of the actions of the solution. Each bad choice for the first action will inevitably lead to a dead-end.

In few problems discarding equivalent sequences is more costly in time. This is the case when there is a great number of actions (see for example Mystery problems and PSR p25 problem). Indeed, ordered 2-sequences and predecessors of actions are computed before the search, and these computations may be costly $O(|A|^2)$. This is also the case when ordering constraints are not efficient, because too many sequences are accepted. For example problems in which most actions interfere will produce soft ordering constraints.

The 2-sequence ordering constraint is generally better than the level ordering constraint. In fact they produce a similar decrease of the number of calls, but in computation time their efficiency is not comparable: testing level ordering is costly since the actions at the preceding steps must be tested, while 2-sequence constraints are easy to test.

Our planner performs generally better than BFHSP, specially for problems with a large number of operators and propositions but short plans, as the mystery domains. On these problems it seems that a breadth-first search will spend a significant amount of time to compute huge layers of states. On the contrary, when there are few actions and long plans, as in Hanoi and FreeCell, BFHSP planners dominate.

Of course, in general, parallel planners perform better than sequential planners. In fact, optimal parallel planners and optimal sequential planners are not comparable, as they do not search for the same optimality, and parallel plans are shorter than sequential plans. However, we have reported the results of SATPLAN since they serve as a reference. Though, when there are many actions or large makespans and the actions interfere (see for example Mystery, Hanoi or some PSR problems), SATPLAN generates large formulas which are hard to solve and overload the memory.

Finally the main advantage of the planner that we present in this paper is its regularity. Computation times increase gradually while the problems become harder.

serie	problem	actions	facts	length	makespan	no ordering		2-seq ordering		level ordering		bfhsp	bfhsp	satplan
						calls	time	calls	time	calls	time	back	forw	
divers	hanoi7	238	94	127		572128	22.1	243271	17.1	243095	17.3	–	–	–
	mystery-x-19	6521	562	6	6	3622	2.2	939	9.8	1087	24.3	736.4	99.7	12.04
	mystery-x-20	7094	575	7	7	14569	5.17	2045	12.9	2555	31.05	1473	315.4	38.1
	xy-world-f10	200	40	10	1	2668	0.04	194314	1.59	429714	3.84	63.7	249.9	0.03
IPC3	driverlog-3-3-6	252	93	13	6	1327338	23.1	458658	15.3	382279	15.7	1.61	664.3	0.07
	zeno-travel-2-5	392	58	11	5	1589396	27.07	474647	11.1	431527	13.03	1.54	298.2	2.41
	zeno-travel-2-6	408	64	15	6	1663353	28.1	718539	18.4	587468	19.6	10.8	773.2	0.87
IPC4	airport-p08	295	443	62	26	1018914	196.6	280170	46.6	290080	55.3	82.5	30.6	1.42
	airport-p14	347	493	60	26	938428	209.2	261852	49.2	272277	59.5	125.6	33.7	1.94
	PSR-p25	9400	58	9	9	4366	12.2	1063	21.5	989	52.3	17.02	63.8	129.2
	PSR-p31	661	63	19	16	357316	21.9	123109	11.6	135203	14.2	3.59	256.4	9.37
	PSR-p46	98	60	34		2051278	26.4	1355273	19.2	1611002	26.8	918.6	66.6	–
IPC5	pipesworld-p04	656	154	11	11	389901	33.7	203215	24.1	218205	29.6	–	631.7	384.7
	pipesworld-p08	2672	204	11	7	5111567	959.3	3333348	857.7	3278869	1244	–	–	273.2
	pipesworld-p21	3272	376	14		1464159	594.3	412206	223.8	480791	285.1	–	–	–
	storage-p11	460	146	17	11	1036518	41.2	340937	21.04	304312	19.3	359.1	85.3	332.1
	storage-p12	690	164	16	9	5148404	256.3	1890755	142.3	1680587	135.7	3833	1479	53.1
	storage-p14	564	203	19	11	8466543	368.5	6976067	421.8	6278393	391.1	4592	5076	294.3
	truck-p02	336	117	17	14	45283	0.8	27058	0.72	30652	0.72	1.26	3.98	21.3
	truck-p03	789	191	20	16	1276518	29.2	686433	23.4	850036	25.7	21.5	406.5	191.2
	truck-p07	1104	275	23		18345815	510.1	10756026	443.8	11679349	422.5	154.6	5837	–

Figure 1: CPU times for different planners on a serie of selected problems (times are in seconds on a Linux computer with a Pentium 3 GHz processor, and 1 GB RAM).

Conclusion and perspectives

We described in this paper a new planner which builds optimal sequential plans. Compared to other optimal sequential planners it seems to be very competitive. Its consistency rules and its decomposition strategies allow to operate backward chaining search or bidirectional search and more generally undirectional search. This planner should be improved implementing backjumping, and we should also experiment a concurrent bidirectional search which could cooperate through valid and invalid states. The lack of termination criterion will be also addressed in future work. Finally valued actions could be handled, with the objective to compute plans of minimal costs. Also, planning with ressource will be a matter of development.

Acknowledgments

We thank the anonymous reviewers for their usefull comments and suggestions.

References

Baioletti, M.; Marcugini, S.; and Milani, A. 2000. Dpplan: An algorithm for fast solutions extraction from a planning graph. In *AIPS*, 13–21.

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

Do, M. B., and Kambhampati, S. 2001. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artif. Intell.* 132(2):151–182.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *AIPS*, 140–149.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In Veloso, M. M., and Kambhampati, S., eds., *AAAI*, 1163–1168. AAAI Press AAAI Press / The MIT Press.

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

Kambhampati, S. 2000. Planning graph as a (dynamic) CSP: Exploiting EBL, DDB and other CSP search techniques in graphplan. *Journal of AI Research* 12(1):1–34.

Kautz, H. A., and Selman, B. 1999. Unifying sat-based and graph-based planning. In *IJCAI*, 318–325.

Lopez, A., and Bacchus, F. 2003. Generalizing graphplan by formulating planning as a CSP. In Gottlob, G., and Walsh, T., eds., *IJCAI*, 954–960. Morgan Kaufmann.

Mackworth, A. 1977. Consistency in networks of relations. In *Artificial Intelligence*, 8:99–118.

Rintanen, J. 1998. A planning algorithm not based on directional search. In *KR*, 617–625.

van Beek, P., and Chen, X. 1999. Cplan: A constraint programming approach to planning. In *AAAI/IAAI*, 585–590.

Vossen, T.; Ball, M.; Lotem, A.; and Nau, D. S. 1999. On the use of integer programming models in ai planning. In *IJCAI*, 304–309.

Zhou, R., and Hansen, E. A. 2004. Breadth-first heuristic search. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *ICAPS*, 92–100. AAAI.