Planning as Satisfiability: Heuristics

Jussi Rintanen

Institute for Integrated and Intelligent Systems, Griffith University, Queensland, Australia

Abstract

Reduction to SAT is a very successful approach to solving hard combinatorial problems in Artificial Intelligence and computer science in general. Most commonly, problem instances reduced to SAT are solved with a general-purpose SAT solver. Although there is the obvious possibility of improving the SAT solving process with application-specific heuristics, this has rarely been done successfully.

In this work we propose a planning-specific variable selection strategy for SAT solving. The strategy is based on generic principles about properties of plans, and its performance with standard planning benchmarks often substantially improves on generic variable selection heuristics, such as VSIDS, and often lifts it to the same level with other search methods such as explicit state-space search with heuristic search algorithms.

Contents

1. Introduction

Translation into SAT, the satisfiability problem of the classical propositional logic, is one of the main approaches to solving the planning problem in AI. The basic idea, first presented by Kautz and Selman [31], is to consider a bounded-horizon planning problem, to represent the values of state variables at every time point as propositional variables, to represent the relation between two consecutive states as a propositional formula, and then to synthesize a propositional formula that is satisfiable if and only if there is a plan of the given bounded length. This idea is closely related to the simulation of nondeterministic polynomial-time Turing machines in Cook's proof of NP-hardness of SAT [14]. Kautz and Selman's idea was considered to be only of theoretical interest until 1996 when algorithms for SAT had developed far enough to make planning with SAT practical and competitive with other search methods [32]. Later, SAT and its extensions have become a major state-space search method in computer-aided verification [5] and in many other areas.

In this work we investigate SAT solving for planning with the conflict-driven clause learning (CDCL) algorithm [37, 3], the currently leading framework for SAT solving for structured problems. Instead of using standard generic CDCL heuristics such as VSIDS [42], we propose planning-specific heuristics which radically differ from generic CDCL heuristics and are based on a simple property all solutions to a planning problem have to satisfy. The work is motivated by the need to better understand why SAT solvers are successful in solving AI planning and other reachability problems, and by the need and opportunity to develop more powerful, problem-specific heuristics for SAT.

Our heuristic chooses action variables that contribute to the goals or subgoals, based on the current partial valuation of the CDCL algorithm, representing a tentative plan and a state sequence corresponding to its execution. The principle is extremely simple: *for a given (sub)goal, choose an action that achieves the (sub)goal and that can be taken at the earliest time in which the (sub)goal can become (and remain)* true. Intuitively, this principle expresses a preference for short and simple plans. After choosing an action, its preconditions become new subgoals for which supporting actions are found in the same way. The principle is easy to implement: start from a goal (or a subgoal), go backwards step by step until a time point in which the goal is *false*, and choose any of the actions that can turn it from *false* to *true* at that time point. If such an action existed in the plan already, perform the procedure recursively with the preconditions of the action as the subgoals.

Interestingly, it turns out that heuristics based on the above principle are often far more effective in finding plans than the sophisticated heuristics used by modern general-purpose SAT solvers. Furthermore, a naïve application of the principle – leading to a depth-first backward chaining planning algorithm inside the CDCL framework – lifts the efficiency of SAT-based planning close to level with the best earlier planners which use other search methods. This is a very significant result, because the currently best state-space search planners, which have their origins in the work of Bonet and Geffner [8], use far more complex heuristics and additional pruning techniques to achieve a comparable performance.

The simplicity and effectiveness of the principle immediately suggests that there are additional heuristics to obtain further efficiency improvements. Instead of finding motivation for such heuristics from

standard benchmarks, we look at generic properties of planning problems and generic structural properties of the search trees generated by the principle. We present heuristics for ordering the new subgoals and for choosing one of the applicable actions, as well as propose a scheme that replaces the pure depth-first backward search by a less directional form of search. For standard benchmark problems in planning, the additional heuristics lift the performance of the new variable selection scheme still further.

We view this work as a step toward developing efficient SAT-based techniques for planning and other related problems such as model-checking [5] and discrete-event systems diagnosis [24]. More advanced heuristics for these applications are likely to also incorporate features from VSIDS, including the computation of *weights* of variables based on their occurrence in recently learned clauses. We believe that the success of the planner developed in this work with the standard planning benchmark problems is more an indication of the simple structure of these benchmarks, and that more challenging problems will need more complex variable selection heuristics. This observation is supported by earlier works that illustrate the scalability of typical planners in solving combinatorially hard problems [46, 54].

The structure of the paper is as follows. Section 2 describes the background of the work in planning with SAT. In Section 3 we present the variable selection scheme for planning, and in Section 4 we propose additional heuristics for it. Section 5 describes the implementation of a planning system that is based on the preceding two sections and our earlier works [50]. In Section 6 we experimentally evaluate the planning system, and in Section 7 we discuss related work before concluding the paper in Section 8.

2. Planning as Satisfiability

2.1. Background

Reduction to the SAT problem was proposed as a way of solving the planning problem in the 1992 paper by Kautz and Selman [31]. At the same time, algorithms for solving the SAT problem were progressing rapidly, and in 1996 Kautz and Selman were able to show their planning system to scale up better than Graphplan [6] and other planning systems of the time [32]. These results were obtained with SAT solvers such as WalkSat [64, 65] and Tableau [15].

The reduction to SAT and the general solution method outlined by Kautz and Selman dominated the SAT approach to planning for the next several years, and became the basis of scientifically and commercially very successful computer-aided verification methods in the form of *bounded model-checking* [5]. In the planning community, however, the focus shifted to heuristic state space search algorithms as proposed by Bonet, Loerincs and Geffner in the HSP planner starting in 1997 [9, 8].

The decreasing interest of planning researchers in SAT at this time can be traced to two factors: the impractically large size of the CNF formulas generated from the standard benchmark problems with the early encoding schemes, and the very high computational cost of completing the satisfiability tests for horizon lengths shorter than the shortest plan.

As proposed by Kautz and Selman, the planners sequentially went through horizon lengths 0, 1, 2, and so on, until they reached a horizon length for which the formula is satisfiable, yielding a plan. Essentially, Kautz and Selman's procedure corresponds to breadth-first search, and these planners proved that the plan that was found had the shortest possible horizon. However, guaranteeing that plans have the shortest possible horizon is unimportant, as the horizon length does not, for commonly used notions of parallel plans, coincide with relevant plan quality measures, such as the sum of action costs. The notion of parallelism also does not correspond to actual temporal concurrency, but the possibility of reordering the parallel actions to a valid sequential plan [50], and therefore should be viewed as a way of inducing a smaller search space.

The unsatisfiability proofs could be avoided by using parallelized search strategies [53]. These often speed up planning by orders of magnitude. At the same time, compact linear-size encodings were proposed. Earlier encodings, such as those based on the planning graphs of Graphplan [6], had a quadratic size, due to the encoding of action mutexes in the most straightforward way as binary clauses. The linear-size encodings largely eliminated the problem of excessive memory consumption, and also otherwise yielded substantial performance improvements [49, 50]. These developments bridged the performance gap between

SAT-based planning and explicit state space search substantially (for standard benchmarks), and reduced the memory consumption so that it was not an obstacle to efficient planning.

Since mid-1990s, there have also been substantial improvements in the performance of algorithm implementations for SAT. The SATZ solver of Li and Anbulagan [34] was influential in the late 1990s, and its implementation techniques were a basis of a very efficient planner with a specialized built-in SAT-style search algorithm [51]. Practically all of the efficient SAT solvers since 2000 have been based on ideas popularized in the zChaff solver [42] which replaced the earlier almost exclusively used Davis-Putnam-Logemann-Loveland procedure [16] by the related conflict-driven clause learning algorithm [37, 3], and introduced the very effective VSIDS heuristic as well as new data structures for very efficient unit propagation. These techniques are also applied in the SAT solver used in this work.

2.2. Formalization of Planning

The classical planning problem involves finding an action sequence from a given initial state to a goal state. The actions are deterministic, which means that an action and the current state determine the successor state uniquely. A state $s : X \to \{0, 1\}$ is a valuation of X, a finite set of *state variables*. In the simplest formalization of planning, actions are pairs $\langle p, e\rangle$ where p and e are consistent sets of propositional literals over X, respectively called *the precondition* and *the effects*. We define $\mathrm{prec}(\langle p, e\rangle) = p$. Actions of this form are known as STRIPS actions for historical reasons. An action $\langle p, e\rangle$ is *executable* in a state s if $s \models p$. For a given state s and an action $\langle p, e\rangle$ executable in s, the unique successor state $s' = exec_{\langle p,e\rangle}(s)$ is determined by $s' \models e$ and $s'(x) = s(x)$ for all $x \in X$ such that x does not occur in e. This means that the effects are true in the successor state, and all state variables not affected by the action retain their values. Given an initial state I, a plan to reach a goal G (a set of literals) is a sequence of actions $a_1, \ldots, a_n$ such that $exec_{a_n}(exec_{a_{n-1}}(\cdots exec_{a_2}(exec_{a_1}(I))\cdots)) \models G$.

2.3. Reduction of Planning to SAT

Kautz and Selman [31] proposed finding plans by a reduction to SAT. The reduction is similar to the reduction of NP Turing machines to SAT in Cook's proof of NP-hardness of SAT [14]. The reduction is parameterized by a horizon length $T \geq 0$. The value of each state variable $x \in X$ in each time point $t \in \{0, \ldots, T\}$ is represented by a propositional variable $x@t$. For each action a and $t \in \{0, \ldots, T-1\}$ we similarly have a propositional variable $a@t$ indicating whether action a is taken at t.

A given set X of state variables, an initial state I, a set A of actions, goals G and a horizon length T is translated into a formula Φ_T such that $\Phi_T \in$ SAT if and only if there is a plan with horizon $0, \ldots, T$. This formula is expressed in terms of the propositional variables $x@0, \ldots, x@T$ for all $x \in X$ and $a@0, \ldots, a@T - 1$ for all $a \in A$. For a given $t \geq 0$, the valuation of $x_1@t, \ldots, x_n@t$, where $X = \{x_1, \ldots, x_n\}$, represents the state at time t. The valuation of all propositional variables represents a state sequence, and the difference between two consecutive states corresponds to taking zero or more actions. The conditions for allowing multiple actions at the same step can be defined in alternative ways [50]. For our purposes it is sufficient that the change from state s to s' by a set E of executed actions satisfies the following three properties: *1)* $s \models p$ for all $\langle p, e\rangle \in E$, *2)* $s' \models e$ for all $\langle p, e\rangle \in E$, and *3)* $s' = exec_{a_n}(exec_{a_{n-1}}(\cdots exec_{a_2}(exec_{a_1}(s))\cdots))$ for some ordering $a_1, \ldots, a_n$ of E. These conditions are satisfied by all main encodings of planning as SAT [61]. The only encodings that do not satisfy these conditions (part 1, specifically) are the relaxed $\exists$-step semantics encoding of Wehrle and Rintanen [70] and the encodings by Ogata et al. [43], which allow the precondition of an action to be supported by parallel actions, instead of the preceding state.

To represent planning as a SAT problem, each action $a = \langle p, e\rangle$ and time point $t \in \{0, \ldots, T-1\}$ is mapped to formulas $a@t \to \bigwedge_{l \in p} l@t$ and $a@t \to \bigwedge_{l \in e} l@(t+1)$.[1] These two formulas respectively correspond to the executability condition and the first part of the definition of successor states. The

[1]For negative literals $l = \neg x$, $l@t$ means $\neg(x@t)$, and for positive literals $l = x$ it means $x@t$. Similarly, we define the valuation $v(l@t)$ for negative literals $l = \neg x$ by $v(l@t) = 1 - v(x@t)$ whenever $v(x@t)$ is defined. The complement $\overline{l}$ of a literal l is defined by $\overline{x} = \neg x$ and $\overline{\neg x} = x$.

second part, about state variables that do not change, is encoded as follows when several actions can be taken in parallel. For each state variable $x \in X$ and time point $t \in \{0, \ldots, T-1\}$ we have

$$x@(t+1) \rightarrow (x@t \vee a_1^x@t \vee \cdots \vee a_n^x@t)$$

where $a_1^x, \ldots, a_n^x$ are all the actions that have x as an effect, for explaining the possible reasons for the truth of $x@(t+1)$, as well as

$$\neg x@(t+1) \rightarrow (\neg x@t \vee a_1^{\neg x}@t \vee \cdots \vee a_m^{\neg x}@t),$$

where $a_1^{\neg x}$, ..., $a_m^{\neg x}$ are all the actions with $\neg x$ as an effect, for explaining the possible reasons for the falsity of $x@(t+1)$. These formulas (often called *the frame axioms*) allow inferring that a state variable does not change if none of the actions changing it is taken.

Additional constraints are usually needed to rule out solutions that don't correspond to any plan because parallel actions cannot be serialized. For example, actions $\langle\{x\}, \{\neg y\}\rangle$ and $\langle\{y\}, \{\neg x\}\rangle$ cannot be made to a valid sequential plan, because taking either action first would falsify the precondition of the other. In our planners, we use the linear-size $\exists$-step semantics encoding of Rintanen et al. [50], which often requires only few or no additional constraints.

There is one more component in efficient SAT encodings of planning, which is logically redundant but usually critical for efficiency: *invariants* [51, 17]. Invariants $l \vee l'$ with two literals express binary dependencies between state variables. Many of the standard planning benchmarks represent many-valued state variables in terms of several Boolean ones, and a typical invariant $\neg x_1 \vee \neg x_2$ says that a many-valued variable x can only have one of the values 1 and 2 at any given time.

For a given set X of state variables, initial state I, set A of actions, goals G and horizon length T, we can compute (in linear time in the product of T and the sum of sizes of X, I, A and G) a formula Φ_T such that $\Phi_T \in$ SAT if and only if there is a plan with horizon $0, \ldots, T$. Φ_T includes the formulas described above, the unit clause $x@0$ if $I(x) = 1$ and $\neg x@0$ if $I(x) = 0$ for $x \in X$, and $l@T$ for all $l \in G$. These formulas are in CNF after trivial rewriting.

A planner can do the tests $\Phi_0 \in$ SAT, $\Phi_1 \in$ SAT, $\Phi_2 \in$ SAT, and so on, sequentially one by one, or it can make several of these tests in parallel (interleave them). For this we will later be using Algorithm B of Rintanen et al. [50] which allocates CPU time to different horizon lengths according to a decreasing geometric series, so that horizon length $t+1$ gets γ times the CPU the horizon length t gets, for some fixed $\gamma \in]0, 1[$. In general, the parallelized strategies can be orders of magnitudes faster than the sequential strategy because they do not need to complete the test $\Phi_t \in$ SAT (finding Φ_t unsatisfiable) before proceeding with the test $\Phi_{t+1} \in$ SAT. Since the unsatisfiability tests, which tend to be far more difficult than determining a formula to be satisfiable, don't need to be completed, it is far more important to efficiently determine satisfiability than unsatisfiability.

2.4. The CDCL Algorithm

In this section we briefly describe the standard conflict-driven clause learning (CDCL) algorithm [37] for solving the SAT problem. This algorithm is the basis of most of the currently leading SAT solvers in the zChaff family [42]. For a detailed overview of the CDCL algorithm and its implementation see standard references [4, 41].

The main loop of the CDCL algorithm (see Fig. 1) chooses an unassigned variable, assigns a truth-value to it, and then performs unit propagation to extend the current valuation v with forced variable assignments that directly follow from the existing valuation by the unit resolution rule. If one of the clauses is falsified, a new clause which would have prevented considering the current valuation is derived and added to the clause set. This new clause is a logical consequence of the original clause set. Then, some of the last assignments are undone, and the assign-infer-learn cycle is repeated. The procedure ends when the empty clause has been learned (no valuation can satisfy the clauses) or a satisfying valuation has been found.

```
 1: procedure CDCL(C)
 2: Initialize v to satisfy all unit clauses in C;
 3: Extend v by unit propagation with C;
 4: level := 0;
 5: do
 6:   if level = 0 and v ⊭ c for some c ∈ C then return false;
 7:   Choose a variable x with v(x) undefined;
 8:   Assign v(x) := 1 or v(x) := 0;
 9:   level := level + 1;
10:   Extend v by unit propagation with C;
11:   if v ⊭ c for some c ∈ C
12:   then
13:     Infer a new clause c and add it to C;
14:     Undo assignments until x so that c is not falsified and decrease level accordingly;
15:   end if
16: while some variable is not assigned in v;
17: return true;
```

Figure 1: Outline of the CDCL algorithm

The selection of the decision variable (line 7) and its value (line 8) can be arbitrary (without compromising the correctness of the algorithm), and can therefore be based on a heuristic. The currently best generic SAT solvers use different variants and successors of the VSIDS heuristic [42]. The heuristic is critical for the efficiency of the CDCL algorithm.

On lines 3 and 10 the standard unit propagation algorithm is run. It infers a forced assignment for a variable x if there is a clause $x \vee l_1 \vee \cdots \vee l_n$ or $\neg x \vee l_1 \vee \cdots \vee l_n$ and $v \models \neg l_1 \wedge \cdots \neg \wedge l_n$. The inference of a new clause on line 13 is the key component of CDCL. The clause will prevent generating the same unsatisfying assignment again, leading to traversing a different part of the search space.

The amount of search performed by the CDCL algorithm can be characterized by the numbers of *decisions* and *conflicts*. The number of decisions is the number of assignments of decision variables, that is, the number of executions of lines 7 and 8. The number of conflicts is the number of executions of line 13. This is usually the number of new clauses learned, although some CDCL implementations may learn multiple clauses from one conflict.

2.4.1. The VSIDS Heuristic

The VSIDS (Variable State Independent Decaying Sum) heuristic [42] for choosing the next decision variable in the CDCL algorithm is based on *weights* of the propositional variables. When the SAT solving process is started, the weight of a variable is initialized to the number of times it occurs in the input clauses. When the CDCL algorithm learns a new clause, the weight of each variable occurring in the clause is increased by one. To decrease the importance of clauses learned earlier, the weights of all variables are divided by some constant at regular time intervals. The VSIDS heuristic chooses as the new decision variable one of the unassigned variables with the maximal weight.

2.4.2. Restarts

An important component in the performance of CDCL is *restarts* [42, 23]. Line 14 makes CDCL a form of backtracking, and long sequences of earlier variable assignments may remain untouched, which often reduces the possibilities of finding a satisfying assignment. To prevent this, the current SAT solvers perform a restart at regular intervals (for example after every 100 conflicts), which means terminating the CDCL algorithm, and starting it from the beginning, but retaining all the learned clauses and the current variable weights. Since the variable weights have changed since the previous restart, the CDCL algorithm will make a different sequence of variable assignments than before, moving the search to a different part of

the search space. For the completeness of CDCL with restarts it is important that the same assignments are not considered repeatedly. Clause deletion to avoid memory overflows and slow down of CDCL [42] risks this, but completeness of CDCL can be theoretically guaranteed by increasing the time interval in which clause deletion is performed. This is what many SAT solvers do, also the one used in our work.

3. The Heuristic

The goal of our work is to present a new way of choosing the decision variables (lines 7 and 8 in the CDCL procedure in Fig. 1) specific to planning. Our proposal only affects the variable selection part, and hence it doesn't affect the correctness or completeness of the CDCL algorithm.

The main challenge in defining a variable selection scheme is its integration in the CDCL algorithm in a productive way. To achieve this, the variable selection depends not only on the initial state, the goals and the actions represented by the input clauses, but also the current state of execution of the CDCL algorithm. The state of the execution is characterized by A) the current set of learned clauses and B) the current (partial) valuation reflecting the decisions (variable assignments) and inferences (with unit propagation) made since the last restart. We have restricted the variable selection to use only part B of the SAT solver state, the current partial valuation.

The variable selection scheme is based on the following observation: each of the goal literals has to be made *true* by an action, and the precondition literals of each such action have to be made *true* by earlier actions (or, alternatively, these literals have to be *true* in the initial state.) Hence, to find the next decision variable for the CDCL algorithm, we find one (sub)goal that is not in the current state of search supported (made *true*) by an action or the initial state.

More concretely, we proceed as follows. The first step in selecting a decision variable is finding the earliest time point at which a (sub)goal (for time t) can become and remain *true*. This is by going backwards from t to time point $t' < t$ in which I) an action making l *true* is taken or II) l is *false* (and l is *true* or *unassigned* thereafter.) The third possibility is that the initial state at time point 0 is reached and l is *true* there, and hence nothing needs to be done. In case I the plan already has an action that makes the subgoal true, and in case II we choose any action that can change l from *false* to *true* between t' and $t' + 1$, and use it as a decision variable.[2] In case I we recursively find support for the literals in the precondition.

The computation is started from scratch at every iteration of the CDCL procedure because a particular support for a (sub)goal, for example the initial state, may become irrelevant because of a later decision, and a different support needs to be found.

When all (sub)goals have a support, the current partial assignment represents a plan. The assignment can be made total by assigning unassigned action variables *false* and unassigned fact variables the value they have in the closest preceding time point with a value.

Notice that the only part of the above scheme for selecting a decision variable that has the flavor of a heuristic is the restriction to the earliest time points in which the (sub)goal can be *true*, corresponding to a preference for short and simple plans.

To find a satisfying assignment for the SAT instance, every (sub)goal has to be made *true*, and the core of our scheme is the focus on the shortest or simplest action sequences for achieving this. This works very well with CDCL because the partial assignments maintained by the CDCL algorithm give useful information about the possibilities of achieving the (sub)goals. Often, when reaching a (sub)goal l seems to be possible at time t but not earlier (meaning that $v \models \overline{l@(t-1)}$ and $l@t$ is unassigned), it is a useful guess that $l@t$ can indeed be made true at that point. And if this is not possible, the CDCL algorithm will often detect this quickly, which leads to trying to make l true at a later time point instead.

[2] Such an action necessarily exists because otherwise l would have to be *false* also at $t' + 1$. This is by the frame axiom for l.

Example 1. *We illustrate the search for an unsupported (sub)goal and the selection of an action with a problem instance with goals a and b and actions $X = \langle\{d\}, \{a\}\rangle$, $Y = \langle\{e\}, \{d\}\rangle$, and $Z = \langle\{\neg c\}, \{b\}\rangle$.*

variable	*0*	*1*	*2*	*3*	*4*	*5*	*6*
a	*0*	*0*			**0**		*1*
b	*0*	*0*	**0**	*1*			*1*
c	*0*	*0*					
d	*0*	*0*	**0**				
e	*1*						

Consider the goal a at time point 6. The latest time point at which a is false is 4, and hence it seems that a could become true at 5 and remain true until 6.

Let's assume that $X@4$ is unassigned, and hence could make a true *at 5. We can then choose $X@4$ as a support for $a@5$, and use $X@5$ as the next decision literal in the CDCL algorithm.*

After $X@5$ is assigned true, we would need support for its precondition d at time point 4. The new subgoal d could change from false *to* true *between time points 2 and 3, and the action Y could be cause of this change. After $Y@2$ has been used as the decision literal, we would need support for its precondition $e@2$, and would determine that no further actions are needed as e is already true in the initial state and can remain true until time point 2.*

For the second top-level goal b, assume that $Z@2$ is assigned true *and hence explains the change of b from* false *to* true *between time points 2 and 3. Since Z's precondition $\neg c$ is satisfied by the initial state, again no further action is required.*

We have marked those timed variables in boldface which change from false *to* true *above.*

The procedure in Figure 2 implements the variable selection scheme as described above. The subprocedure terminate$(X, a@t)$ implements a termination condition, which in our baseline case is terminate$(Z, a@t)$ iff $|Z| = 1$, i.e. the first action found is used as the next decision variable. The subprocedure cleanup$(X, a@t)$ decides whether (after identifying action $a@t$) to return the whole set Z or something less. The baseline implementation returns Z as is, i.e. cleanup$(Z, a@t) = Z$. In Section 4 we consider alternative implementations of these subprocedures. Given the goal G, the actions A, the horizon length T, and a partial valuation v of the propositional variables representing the planning problem with horizon length T, the procedure call support(G, A, T, v) will return a set of candidate decision variables.

The procedure starts with inserting all goal literals in G to a priority queue. The ordering of the literals in the queue determines the order in which candidate actions are generated. In the baseline version of the variable selection scheme we define $<_0$ as the empty relation, so that the queue acts as a stack (last in, first out.) Later we will consider more informed orderings. The priority queue as a stack together with the computation of only one action for the next decision variable force the CDCL algorithm to do a form of backward chaining depth-first.

On line 7 the next (sub)goal literal $l@t$ is taken from the queue. The loop between lines 10 and 24 identifies an action that supports or could support $l@t$. It goes from time point $t-1$ step by step to earlier time points, until it finds an action that supports $l@t$ (line 11) or the earliest time point t' at which $l@t$ can become and remain true (16).

If an action was found, the search must continue with finding support for the preconditions of the action. For this purpose the preconditions are inserted in the priority queue on line 13.

If the earliest time t' in which l can be supported was found without an action, then one of the actions that can make $l@t'$ true is chosen on line 17. This action is added to the set Z on line 18, and the set is returned if the termination condition is met. If the termination condition is not yet met, the preconditions of the action are inserted in the priority queue on line 20.

The choice of a on line 17 of Fig. 2 is arbitrary, but it should be fixed for example by choosing the first $a \in A$ that satisfied the two conditions in some fixed ordering. Intuitively, this is important to avoid losing focus in the CDCL search. Similarly the ordering of (sub)goal literals in the priority queue in situations in which $<$ does not strictly order one before the other is arbitrary, but should be similarly fixed for the same reason.

```
1:  procedure support(G, A, T, v)
2:  Unmark all literals;
3:  Empty the priority queue;
4:  for all l ∈ G do insert l@T into the queue according to < and mark it;
5:  Z := ∅;
6:  while the queue is non-empty do
7:    Pop l@t from the queue;                          (* Take one (sub)goal. *)
8:    t' := t − 1;
9:    found := 0;
10:   repeat
11:     if v(a@t') = 1 for some a ∈ A with l ∈ eff(a)
12:     then                                           (* The subgoal is already supported. *)
13:       for all unmarked l' ∈ prec(a) do
14:         insert l'@t' into the queue according to < and mark it;
15:       found := 1;
16:     else if v(l@t') = 0 then                       (* Earliest time it can be made true *)
17:       a := any a ∈ A such that l ∈ eff(a) and v(a@t') ≠ 0;
18:       Z := Z ∪ {a@t'};
19:       if terminate(Z, a@t') then return cleanup(Z, a@t');
20:       for all unmarked l' ∈ prec(a) do
21:         insert l'@t' into the queue according to < and mark it;
22:       found := 1;
23:     t' := t' − 1;
24:   until found = 1 or t' < 0;
25: end while
26: return Z;
```

Figure 2: Computation of supports for (sub)goals

```
1:  S := support(G, A, T, v);
2:  if S ≠ ∅ then v(a@t) := 1 for a@t = choose(S);          (* Found an action. *)
3:  else
4:    if there are unassigned x@t for x ∈ X and t ∈ {1, ..., T}
5:    then v(x@t) := v(x@(t − 1)) for any unassigned x@t with minimal t
6:    else v(a@t) := 0 for any a ∈ A and t ∈ {0, ..., T − 1} with a@t unassigned;
```

Figure 3: Variable selection for planning with the CDCL algorithm

```
1:  procedure choose_random(S)
2:  return randomly chosen element of S;
3:
4:  procedure choose_weighted(S)
5:  return an element of S with the highest VSIDS weight, with ties broken randomly;
```

Figure 4: Procedures from selecting one of several candidate actions

3.1. Integration in the CDCL Algorithm

The procedure in Fig. 2 is the main component of the variable selection scheme for CDCL given in Fig. 3, in which an action is chosen as the next decision variable for the CDCL algorithm if one is available. If several are available, one is chosen either randomly (formalized as the procedure $\text{choose}_{random}(S)$ in Figure 4) or according to the weights as calculated for VSIDS-style heuristics when learning a new clause (formalized as the procedure $\text{choose}_{weighted}(S)$ in Figure 4). The weight parameter of an action occurrence $a@t$ is increased for its every occurrence in a learned clause, and all the weights are divided by two in regular intervals, after every 32 conflicts.

If no actions are available, all goals and subgoals are already supported. The current valuation typically is still not complete, and it is completed by assigning unassigned fact variables the value they have in the predecessor state (line 5) and assigning unassigned action variables the value *false* (line 6). The code in Fig. 3 replaces VSIDS as the variable selection heuristic in the CDCL algorithm of Fig. 1. This is by removing lines 7 and 8 and replacing them by the code in Fig. 3. Note that some actions are inferred by unit propagation on line 10 in the CDCL algorithm, and these actions are later handled indistinguishably from actions chosen by the heuristic.

3.2. Complexity of the Variable Selection Algorithm

If there are n state variables and the horizon length is T, then there are nT variables that represent state variables at different time points. Since each literal is inserted into the priority queue at most once, the algorithm does the outermost iteration on line 6 for each goal or subgoal at most once, and hence at most nT times in total. The number of iterations of the inner loop starting on line 10 is consequently bounded by nT^2.

The actual runtime of the algorithm is usually much lower than the above upper bound. A better approximation for the number of iterations of the outer loop is the number of goals and the number of preconditions in the actions in the plan that is eventually found. In practice, the runtime of the CDCL algorithm with our heuristic is still strongly dominated by unit propagation, similarly to CDCL with VSIDS, and computing the heuristic takes somewhere between 5 and 30 per cents of the total SAT solving time, depending on the properties of the problem instance.

4. Refinements to the Heuristic

The variable selection scheme from Section 3 alone, without any further heuristics, leads to a powerful planner. However, experience from SAT solvers and from the application of SAT solving to planning specifically [51] suggests that the fixed goal orderings and the strict backward chaining depth-first search

are not the best possible way of using CDCL. In this section we consider three strategies to more effectively take advantage of the strengths of the CDCL algorithm.

First, we will present a goal ordering heuristic for controlling the priority queue. The order in which the algorithm in Fig. 2 encounters actions directly determines the ordering in which variables are assigned in the CDCL algorithm, assuming that only the first action found is returned.

Second, we can use a heuristic for choosing which action to use to achieve a subgoal, instead of choosing an arbitrary action.

Third, the search with strict backward chaining will be relaxed. Backward chaining means selecting an action with an effect x given a goal x, and taking the preconditions of the action as new goals, for which further actions are chosen. The search with backward chaining proceeds step by step toward earlier time points (until some form of backtracking takes place.) With CDCL and other SAT algorithms, the search does not have to be directional in this way, and actions less directly supporting the current (sub)goals could be chosen, arbitrarily many time points and actions earlier. The algorithm in Fig. 2 can be forced to compute a complete set of candidate actions for supporting all goals and subgoals and their preconditions, but randomly choosing one action from this set is not useful, and we need a more selective way of using them.

Next we will consider these three possible areas of improvement, and in each case propose a modification to the basic variable selection scheme which in Section 6 will be shown to lead to substantial performance improvements.

4.1. Goal Ordering

Using the priority queue as a stack, as in the baseline implementation of the variable selection scheme in Section 3, leads to depth-first search in which the traversal order of the children of a node is arbitrarily determined by the order in which they are inserted in the queue.

As an alternative to using the queue as a stack, we considered the ordering $<_v$ which orders (sub)goals as $l_1@t_1 <_v l_2@t_2$ if and only if $m_v(l_1@t_1) < m_v(l_2@t_2)$, where $m_v(l@t)$ is defined as the maximal $t' < t$ such that $v(l@t') \neq 1$. Here $v(l@t') \neq 1$ includes the case that $v(l@t')$ is unassigned. According to this ordering, l gets a higher priority if it must have been made *true* earlier than other subgoals. The most likely plan first makes l *true*, followed by the other subgoals. Intuitively, this measure is an indicator of the relative ordering of the actions establishing different preconditions of a given action.

We tried out some other similar simple orderings, but experimentally they did not improve the planner performance.

A key property of the m_v measure is that for every goal or subgoal $l@t$, the new subgoals $l_1@t - 1, \ldots, l_n@t - 1$ all have a higher priority than their parent $l@t$. This will still lead to depth-first search, but the ordering of the child nodes will be more informed.

4.2. Choice of Action

On line 17 in the algorithm in Figure 2 the choice of the action is left open. For each action $a@t$ we calculate a score that is the number of time points following t that the action a could be taken (that is, the number of time points $t' > t$ such that the variable $a@t$ is unassigned.) Then we choose an action occurrence $a@t$ with a minimal score. Intuitively, this score measures how constrained the candidate actions are. More constrained actions are more likely to lead to further inferences or an early detection of a contradiction for the current partial plan.

4.3. Computation of Several Actions

To make the plan search less directional, we experimented with computing a set S of some fixed number N of actions and randomly choosing one $a@t \in S$. In the framework of the algorithm in Fig. 2 this means defining terminate$(S, a@t)$ to return *true* when $|S| = N$. The initial experiments seemed very promising in solving some of the difficult problems much faster. However, the overall improvement was relatively small, and it very surprisingly peaked at $N = 2$.

```
 1: procedure terminate_0(S, a@t)
 2: if |S| = 1 return true;
 3: return false;
 4:
 5: procedure cleanup_0(S, a@t)
 6: return S;
 7:
 8: procedure terminate_1(S, a@t)
 9: if |S| ≥ 10 or t ≥bound return true;
10: return false;
11:
12: procedure cleanup_1(S, a@t)
13: return S\{a@t};
```

Figure 5: Different subprocedures of the selection scheme

What happened is the following. For a given top-level goal $l \in G$, several of the first actions that were chosen supported the goals. However, after everything needed to support l was included, the computation continued from the *next unsupported top-level goal*. Consequently, at the final stages of finding support for a top-level goal we would be, in many cases, selecting supporting actions for other top-level goals, which distracts from finding support for l. With $N = 2$ the distraction is small enough to not outweigh the benefits of considering more than one action.

This analysis led us to a second variant, which proved to be very powerful. We record the time-stamp t of the first action found. Then we continue finding up to N actions, but *stop and exit* if the time-stamp of a would-be candidate action is $\geq t$. This means defining terminate$(S, a@t)$ as *true* if $|S| = N$ or $t >$ bound, where *bound* is initialized right after line 17 by **if** $Z = \emptyset$ **then** bound $:= t$, and cleanup$(S, a@t) = S\backslash\{a@t\}$ if $t >$ bound and S otherwise. With this variant we obtained a substantial overall improvement with higher N. In our experiments we used $N = 40$.

4.4. Variants of Our Planner

Later in Section 6 we refer to different variants of our planner as follows, based on different implementations of terminate$(S, a@t)$ and cleanup$(S, a@t)$ from Figure 5, and either the trivial ordering $<_0$ of the priority queue leading to a stack behavior or the more informed ordering $<_v$ from Section 4.1. The base planner uses the uninformed selection of subgoals and does backward chaining. Backward chaining is enforced by only computing one action that supports the current subgoal. The less directed form of search is obtained by computing several actions, and then choosing one of them randomly with choose$_{random}(S)$ or according to the VSIDS-style weights with choose$_{weighted}(S)$ (Figure 4). The informed action selection, based on a heuristic, is as described in Section 4.2.

The different features with which our planner can be configured are listed in Table 1. The feature a denotes the informed action selection from Section 4.2, the feature g the ordering of goals from Section 4.1, and the features m and w the relaxed action selection from a set of actions respectively by random choice and by VSIDS weights. The two choices with and without a, the two choices with and without g, and the three choices with m or w or neither, induce 12 different configurations in which the new heuristic can be used. Our planner with the *agw* configuration is called Mp, and our planner configured to use the generic SAT heuristic VSIDS is called M. Essentially, M is an efficient implementation of the planner described in our work from 2006 [50]. The 12 different configurations of the planner are experimentally compared in Section 6.6.1.

4.5. Discussion

The good performance of the fixed and uninformed variable selection is due to its focus on a particular action sequence. Any diversion from a previously tried sequence is a consequence of the clauses learned

feature	termination	goal ordering	action choice	search
base	$\text{terminate}_0, \text{cleanup}_0$	$<_0$	arbitrary	backward chaining
`a--`			informed	
`-g-`		$<_v$		
`--m`	$\text{terminate}_1, \text{cleanup}_1$			non-directional, $\text{choose}_{random}(S)$
`--w`	$\text{terminate}_1, \text{cleanup}_1$			non-directional, $\text{choose}_{weighted}(S)$

Table 1: List of features of different planner configurations

with CDCL. This maximizes the utility of learned clauses, but also leads to the possibility of getting stuck in a part of the search space void of solutions. A remedy to this problem in current SAT solvers is restarts [42]. However, with deterministic search and without VSIDS-style variable (or action) weighting mechanism, restarts make no difference, as the assignment right before the restart would necessarily be generated right after the restart. In SAT algorithms that preceded VSIDS, a small amount of randomization in the selection of the decision variable was used to avoid generating the same assignments repeatedly [22]. However, too large diversion from the previous action sequences makes it impossible to benefit from the clauses learned with CDCL. Hence the key problem is finding a balance between focus to recently traversed parts of the search space and pursuing other possibilities.

The flexible depth-first style search from Section 4.3 provides a balance between focus and variation. The candidate actions all contribute to one specific way of supporting the top-level goals, but because they often don't exactly correspond to an actual plan (except for at the very last stages of the search), varying the order in which they are considered seems to be an effective way of probing the "mistakes" they contain.

5. Implementation

The implementation of the planner uses the techniques introduced in our earlier works, including the compact linear-size encoding of the ∃-step semantics [50] and the parallel evaluation strategy implemented by Algorithm B [53, 50]. Next we describe the different components of the planner in detail.

5.1. Encoding

The ∃-step encoding we use in our planners differs from the traditional encodings with respect to parallelism. Traditional encodings (called ∀-step encodings in our earlier works [50]) allow a set of actions in parallel if imposing any total ordering on them results in an executable action sequence. A sufficient condition for this is that the actions don't interfere: no action disables a parallel action or affects its (conditional) effects. The traditional way to encode this condition is to use action mutexes $\neg a_1@t \vee \neg a_2@t$ to state that interfering actions a_1 and a_2 cannot be taken simultaneously. In the worst case, the number of these mutexes is quadratic in the number of actions, and this is a main reason for the very large size of traditional encodings.

The ∃-step plans relax ∀-step plans by only requiring that there is *at least one* total ordering of the parallel actions. There are several alternative ways of implementing this substantially more relaxed condition [50]. The simplest modification to traditional encodings is to only change the action mutexes. Instead of requiring that the disables/affects relation restricted to the simultaneous actions is empty, it is only required that this relation is acyclic [50]. There is a simple encoding for this that is linear in the size of the actions' effects [50], based on imposing a fixed total ordering on the actions and requiring that no action disables/affects a later action.

Further, for almost all of the standard planning competition benchmarks the disabling/affects relation restricted to sets of actions without mutually contradicting preconditions or effects can contain only small cycles or no cycles at all: this can be shown by computing the strongly connected components (SCC) of a *disabling* graph [50]. Any cycle must be contained in an SCC. SCCs of size 1 cannot be involved in a

cycle, and if all SCCs are of size 1, no action mutexes are needed at all. Hence many and sometimes all of the cycles are eliminated already because sets of actions with mutually contradicting preconditions or effects cannot be taken in parallel anyway.

Also the general linear-size encoding scheme for action mutexes is improved when it can be restricted to small SCCs rather than the whole action set. For small SCCs some of the auxiliary variables required in the general for of the encoding can be easily eliminated. Also, our planner can choose between the general linear-size encoding and the trivial worst-case quadratic explicit encoding of action mutexes which does not require auxiliary variables [50]. The latter can be better if the SCC is small and many pairs of actions have mutually contradictory preconditions or effects.

In summary, the two benefits of $\exists$-step encodings over the traditional $\forall$-step encodings are that more actions are allowed in parallel, reducing the horizon lengths and therefore speeding up search, and the number and complexity of action mutexes is reduced and they are often not needed at all, further speeding up search and reducing memory requirements.

The computation of the disabling graphs is one of two time-consuming parts of the front-end of our planner, when the number of actions is tens or hundreds of thousands. This is because the number of arcs in the graphs can be quadratic in the number of actions (nodes) in the worst case. For 80 per cent of the planning competition instances (which are experimented with in Section 6.6) the graphs are computed in less than 2 seconds, and for 16 instances it took more than 60 seconds. This computation is highly optimized. Naïve implementations would slow down the planner considerably. For example, instead of constructing the disabling graph explicitly before running Tarjan's strongly connected components algorithm, it is much more efficient to generate on the fly only those arcs that are actually followed by Tarjan's algorithm. Explicit generation of the graph would unnecessarily spend substantial amounts of time determining existence of irrelevant arcs. Also, we used compact data structures which increase locality of memory references and decrease the number of cache misses.

5.2. Invariants

An important part of efficient SAT encodings of planning is *invariants*, facts that hold in the initial state and will continue to hold after any number of actions have been taken. Computing invariants is the second part of the planner's front-end sometimes with a high overhead.

The identification of invariants is important for many types of planning problems represented in languages like PDDL and STRIPS that only support Boolean state variables. What would naturally be a many-valued state variable in higher level languages is often represented as several dependent Boolean variables in PDDL and STRIPS. Recognizing and explicitly representing these dependencies, that an n-valued state variable cannot have two different values simultaneously, is critical for efficiently solving most of the standard benchmark problems with SAT. As in most works on planning, we restrict to 2-literal invariants $l \vee l'$ which are sufficient for representing the most important variable dependencies.

We used a powerful yet simple fixpoint algorithm for computing invariants [56]. Our implementation works with the grounded problem instance, and it slows down when the number of ground actions increases to tens or hundreds of thousands. The computation of invariants for 90 per cent of the planning competition instances in Section 6.6 takes less than 2 seconds, and for 42 instances it takes over 60 seconds. For the largest instances of AIRPORT/ADL and VISITALL it takes several minutes. Similarly to disabling graphs, the invariant computation is highly optimized to minimize cache misses.

Our planner simplifies action sets based on information given by invariants. Actions that have a precondition that contradicts an invariant and therefore cannot be a part of a valid plan, are eliminated. If the literal l is an invariant, its occurrences are eliminated from all actions. Pairs of literals l and l' such that both $l \vee \overline{l'}$ and $\overline{l} \vee l'$ are invariants (which is equivalent to $l \leftrightarrow l'$) always have the same value. All occurrences of one of the literals are replaced by the other literal.

5.3. SAT Solver

Unlike in the experiments described in the earlier article [50], the new planner uses our own SAT solver implementation, with data structures supporting the interleaved solution of several SAT instances

for different horizon lengths of the same problem instance inside one process and thread. The solver goes through the instances in a round-robin manner, switching from instance to instance at every restart. The solver's clause database is shared by all of the SAT instances, and also some other data structures are shared, including the binary input clauses which typically strongly dominate the size of the clause sets that represent planning problems. As there is a copy of the same binary clauses for representing actions and invariants for every time point, our SAT solver represents the clauses in a parameterized form, with time as the parameter [59]. This often decreases memory consumption substantially, and reduces the number of cache misses when accessing the clauses.

The CDCL implementation in our SAT solver is conventional. It includes the VSIDS heuristic as an alternative heuristic, the phase selection heuristic from RSAT to enhance VSIDS [45], and a watched literal implementation of unit propagation as in the zChaff solver [42]. We tried different clause learning schemes and decided to use the Last UIP scheme as it seems equally good as the more commonly used First UIP scheme [41]. We make a restart after every 60 learned clauses.

The computation of the new heuristic is relatively expensive, but not substantially more so than VSIDS. For a small collection of hard planning problems for which a substantial amount of search was needed, the SAT solver spent 59.76 per cent of the time doing unit propagation, 22.54 per cent of the time computing the heuristic, and the remaining 17.7 per cent with the rest of the CDCL algorithm, including learning clauses and maintaining the clause set. These percentages were measured with the *gprof* profiler with code instrumented by the *gcc* compiler. The code was compiled without function inlining to enable measurement by function, which may distort the relative percentages in comparison to fully optimized code.

The main difference between our SAT solver and the best generic SAT solvers is that we don't use a preprocessor to logically simplify the clause sets. In our experiments, we found the generic preprocessing techniques to be too slow for the very large clause sets obtained from planning, to the extent that they hamper efficient planning. Preprocessing can reduce the size of the search space exponentially, but for the kind of very large SAT problems experimented with in Section 6.6 and the relatively short runtimes (30 minutes), the exponential reductions don't materialize. We will comment on the relative efficiency of our SAT solver with respect to generic SAT solvers in Section 6.4.

5.4. Top-Level Planning Procedure

The top-level procedure of our planner solves SAT instances corresponding to the planning problem for different plan lengths. The planner implements a number of alternative strategies.

The traditional sequential strategy, first presented by Kautz and Selman [31], solves the SAT problem for horizon length 1 first, and if the formula is unsatisfiable, it continues with horizon lengths 2, 3 and so on. This procedure corresponds to breadth-first search, in which all action sequences of length $n-1$ are considered before proceeding with sequences of length n. It is often very ineffective because of the hardness of the unsatisfiable formulae corresponding to horizon lengths below the shortest plan [53].

In all of the experiments reported later (unless otherwise stated), we use a more effective strategy formalized as algorithm B by Rintanen [53], with $\gamma = 0.9$. The algorithm interleaves the solution of several horizon lengths, and its important property is that the satisfiability test for a formula for horizon length n can be started (and completed) before the tests of unsatisfiable formulas for lengths $< n$ have been completed. This way the planner does not get stuck with very hard unsatisfiable formulae. Rintanen's algorithm B allocates CPU to formulae $\Phi_0, \Phi_1, \Phi_2, \ldots$ representing different horizon lengths according to a geometric series: formula Φ_n gets CPU proportional to γ^n where γ is a constant satisfying $0 < \gamma < 1$. The SAT problem for the shortest active horizon length gets $1-\gamma$ per cent of the CPU: with $\gamma = 0.9$ this is 10 per cent and with $\gamma = 0.5$ it is 50 per cent. Conceptually the algorithm considers an infinite number of horizon lengths, but to make the algorithm practical, our planner is solving, at any given time, the SAT problems of at most 20 horizon lengths. As some instances are shown unsatisfiable, the solver is started for longer horizon lengths. Our planners consider horizon lengths 0, 5, 10 and so on.

Rintanen [53] has shown that algorithm B can be – in comparison to the traditional sequential procedure – arbitrarily much faster, and it is never slower by more than a factor of $\frac{1}{1-\gamma}$.

Other top-level algorithms exist, including Rintanen's algorithm A [53, 71] which equally splits CPU to a fixed number of SAT solving processes, and Streeter and Smith's algorithms [67] which perform a form of a binary search to identify the satisfiable formula with the shortest horizon length. Streeter and Smith's algorithms require as input an upper bound on the horizon length, which is generally available in scheduling problems but not in planning. Rintanen's algorithm A is often comparable to algorithm B, but overall appears somewhat worse.

5.5. Conjunctive Conditional Effects

Our planner accepts the general (non-temporal) PDDL language [39]. The heuristics can be extended to cover the full language, with conditional effects and disjunctive conditions [58]. In the experiments reported later, we include problems with conditional effects, but no disjunction. The extension to the heuristic required to handle conditional effects is simple, and the further extension to cover disjunction is more complicated [58].

A formula is *conjunctive* if it is a conjunction of one or more literals and constants $\top$ or $\bot$. An action is *conjunctive*, if its precondition is conjunctive and for every *conditional effect* $\phi \rhd l$ the condition ϕ is conjunctive. The effect l of the conditional effect is made true if and only if the condition ϕ is true when taking the action.

Instead of considering literals to be achieved by actions, we consider them to be achieved by conditional effects $\phi \rhd l$ of actions, with $\top \rhd l$ for handling unconditional effects. When translating a planning problem into a propositional formula, we introduce a propositional variable for every conditional effect (effects $\phi \rhd l_1$ and $\phi \rhd l_2$ of one action may use the same propositional variable, because these conditional effects must always take place together.)

For example, the action $\langle \phi, \{x \rhd y, z \rhd w\}\rangle$ is translated into a formula with the two auxiliary variables u_1 and u_2 for the two conditional effects as follows.

$$
\begin{aligned}
&a@t \rightarrow \phi@t\\
&a@t \wedge x@t \rightarrow u_1@t\\
&u_1@t \rightarrow y@(t+1)\\
&u_1@t \rightarrow x@t\\
&u_1@t \rightarrow a@t
\end{aligned}
$$

$$
\begin{aligned}
&a@t \wedge z@t \rightarrow u_2@t\\
&u_2@t \rightarrow w@(t+1)\\
&u_2@t \rightarrow z@t\\
&u_2@t \rightarrow a@t
\end{aligned}
$$

With the u_i variables the frame axioms

$$(y@t \wedge \neg y@(t+1)) \rightarrow u_1@t \vee \cdots$$

can be trivially turned to clauses.

The propositional variables for the conditional effects are handled in our heuristic exactly like the propositional variables for actions.

6. Evaluation

In the experimental part of this work, we make a comparison between the traditional CDCL heuristics such as VSIDS and the new heuristics proposed in this work, and between our SAT-based planners and planners that use other search methods, including explicit state-space search [8] and stochastic search in the space of plans [18].

Our planner with VSIDS as the decision heuristic, corresponding to our earlier work [50], has already been shown to dramatically outperform planners with the BLACKBOX architecture [33] on the standard benchmark sets, so we don't include these planners in the comparison.[3]

As the goal of the work is to show that the new heuristics are competitive for the standard benchmarks from the planning competition, the main focus of the comparison to other planners and to SAT-based planning with generic heuristics is with the best configuration of our new heuristics (agw), as determined in Section 6.6.1. This is the planner we call Mp.

In Sections 6.5.1, 6.5.2 and 6.5.3 we compare Mp to M, the variant of our planner that uses the standard VSIDS decision heuristic, with a number of problem classes for which standard SAT solvers are known to perform particularly well. In Section 6.6 we shift focus to the planning competition benchmarks, first showing that our new variable selection heuristic is outperformed by VSIDS with *unsatisfiable* formulas. Then we show, however, that the heuristics' behavior with *satisfiable* formulas is far more critical for the planners' performance with the planning competition benchmarks, to the extent that the new heuristic lifts the efficiency of SAT as a planning method to the same level with best earlier planners (Section 6.6).

6.1. Test Material

We use problem instances from 3 different categories of planning problems. Next we describe these categories in more detail, pointing out some of the limitations of each.

1. We test more than 1600 problem instances from the biennial planning competitions from 1998 to 2008, and the 2011 competition [29], which represent small to very large planning problems, with the largest instances having thousands of state variables and hundreds of thousands of actions, and still often solved quickly by many planners. A detailed list of the benchmark domains is given in Table 5 on page 31. These are the problems many works on classical planning focus on, including ones evaluating and comparing different planners.
   The planning competition domains represent a wide range of mostly simplified planning scenarios from transportation, autonomous systems, control of networked systems such as oil pipelines, as well as a number of less serious scenarios such as stacking and unstacking blocks, mixing cocktails, or parking cars.
   The instance sets available and commonly used by the planning community could be more informative for planner evaluation. First, for many of the domains, all or most of the planners solve all the instances quickly, often in a matter of seconds. Little or no information about the planners' scalability and asymptotic behavior can be obtained in these cases. Second, in the 2011 competition, for many domains the instances are relatively hard, some planners not solving any or only few instances, and for some of the domains all instances are of roughly the same difficulty. In these cases we only obtain qualitative information, that some planners are equally strong or (at least somewhat) stronger than some others for instances of a given difficulty level, but it is not possible to quantify this difference further or asymptotically.
2. Of combinatorially harder planning problems, which earlier have been best solved by generic SAT solvers, we used problems obtained through translations from hard combinatorial graph and other problems into planning [46]. We use the problem sets made available by Bonet on his web page.
3. Other classes of small but hard problems are those obtained by sampling from the space of all planning problems, with a sampling distribution experimentally determined to produce hard problems. These problems were first considered by Bylander [11]. We use elaborations of Bylander's models by Rintanen [54], as well as a newer model that only produces solvable instances [60].
   For the first class of problems, we use a sequence of problem instances that covers the phase transition region from easy to hard to easy instances, for a fixed number of state variables and a varying number of actions.

[3] None of the recent published works on planning with SAT make an experimental comparison to other search methods because of the large performance gap between BLACKBOX style planners and recent state-space search planners such as LAMA [48].

For the second class of problems, we go through a sequence of different problem sizes, increasing the number of both actions and state variables linearly while keeping the parameters affecting the relative difficulty constant.

We describe the instances from each of these categories in more detail in Sections 6.6, 6.5.1 as well as 6.5.2 and 6.5.3, respectively. The main point of comparison for the other search paradigms is the benchmarks from the planning competitions, for which SAT earlier was not very competitive. For the combinatorially harder problems we demonstrate the trade-off represented by our new heuristics: the new heuristic is generally between SAT-based planning with VSIDS as implemented in our M planner, and the planners that don't use SAT. M is in general the strongest planner with these problems.

6.2. Other Planners Evaluated

In addition to SAT-based planning with general-purpose heuristics, we make a comparison to planners that don't use SAT. The most competitive planners are ones from the HSP family of planners [8], which use explicit state space search, and the LPG-td planner [18] which uses two search algorithms, one doing search in the space of partial plans and the other doing explicit state space search.

A distinction between planners is whether they – like LPG-td – use a *portfolio* of algorithms or only use one algorithm. To give more depth to our evaluation, we look at the individual components of some of the earlier portfolio-based planners, as well as consider the impact of our new planner when used as a part of a portfolio. Algorithm portfolios [28, 20, 21] have been recognized as an important approach to solving hard combinatorial problems when different individual algorithms have complementary strengths and none of the individual algorithms alone is very strong. Portfolios can be used in different ways, either choosing one algorithm based on the characteristics of the problem instance at hand, or running several of the algorithms in parallel, or according to a schedule.

The planning community has used small portfolios, typically consisting of two algorithms, and the constituent algorithms have been selected by trial and error based on their performance with the standard benchmark sets. The first planner to use portfolios was BLACKBOX [33]. It used hand-crafted schedules which determined which SAT solvers are run in which order and for how long. Later, FF and LPG-td used two-algorithm portfolios, with the second algorithm run after the first algorithm had terminated without finding a plan, according to a termination criterion. Lots of other portfolios are possible, obtained by combining any collection of two or more search algorithms. Some of them are discussed in Section 6.7.

In our experimental study, we compare our SAT-based planners to the following planners.

1. HSP is the original heuristic state-space search planner by Bonet and Geffner [9, 8]. It implements a number of search algorithms and heuristics based on delete relaxations.
   HSP restricts to the STRIPS subset of (non-temporal) PDDL without disjunctive conditions or conditional effects.
   We used HSP version 2.0 with forward search (explicit state-space search), the additive *sum* heuristic, and the best-first search algorithm. We used the `-w 5.0` option as suggested by Bonet and Geffner, to make the search more greedy.
2. FF [26] is a planner that uses a 2-algorithm portfolio, adding an incomplete local search phase before a HSP-style complete search. The first phase does a hill-climbing search with the generation of successor states restricted to a subset generated by *helpful* actions to minimize the time spent evaluating the heuristic value of states, and with exhaustive breadth-first search to escape plateaus and local minima. FF's first phase also uses a goal agenda mechanism which is critical for solving many of the Blocks World instances, some Airport instances, but otherwise has little or no impact. The second phase is essentially HSP with a heuristic similarly based on delete relaxations but with a different method for counting the actions required to reach the goals, and with a similar performance. The second phase is started after the first phase cannot escape a local minimum.
   FF's parser did not parse half of the instances of the 2008 TRUCK domain because of a stack overflow caused by a recursive grammar rule, but these instances are already beyond FF's reach, so this issue does not affect the results of the experiments.

3. LPG-td [18] is a 2-algorithm portfolio similarly to FF. Its first stage is stochastic local search in the space of partial plans. The second stage is FF's HSP-style best-first search. Of the algorithms we test, the first phase of LPG-td is the only one that uses neither SAT nor explicit state-space search, and its heuristics are radically different from the ones used by the other planners.
   Similarly to HSP, LPG-td is restricted to STRIPS.
   We ran LPG-td with the default settings and a preference for low runtimes rather than small plans. When testing the first phase only with the `-nobestfirst` option, we changed the `-restarts` setting from the low default value to 1000 to make use of all of the available 1800 seconds rather than giving up too early. The LGP-td binary that was available to us limited to 10000 actions, not allowing to run the planner with a number of the largest instances, in particular those of LOGISTICS, which the planner would probably have solved quickly. Similarly, a hard limit on the number of goal literals in the binary we had prevented LGP-td from solving a number of VISITALL instances it otherwise solved quickly.
   LPG-td incorrectly claims the unsolvability of 5 instances of the PARCPRINTER domain, which it otherwise solves easily.
4. YAHSP [69] uses FF's heuristic and a best-first search algorithm with preference to actions that are *helpful* according to the FF definition. YAHSP also introduces shortcuts to the search space obtained from prefixes of relaxed plans computed as a part of the heuristic. Unlike FF and LPG-td, YAHSP consists of one phase only. Similarly to HSP and LPG-td, YAHSP only supports STRIPS. We ran the planner with its default settings. Equality is incorrectly implemented in YAHSP's front-end, requiring small modifications in the TPP and SCANALYZER domain files: See Section 6.6 for details.
5. LAMA [48] combines FF's heuristic with another heuristic (landmarks), and it has a scheme for preferring successor states reached by helpful actions that differs from YAHSP's. Unlike the other planners, LAMA's preprocessor constructs a many-valued representation from the Boolean PDDL representation, and thus has a more compact higher-level representation to work with than the other planners. LAMA has one phase only, and a main difference to YAHSP is – in addition to the use of an aggregate of heuristics – the lack of the shortcut mechanism. The purpose of LAMA's landmark heuristic is to improve the quality of plans and it has little impact on its performance and scalability otherwise [48].
   LAMA, similarly to M, Mp and FF, supports the PDDL language with conditional effects (ADL).
   We ran two variants of LAMA, the 2008 competition version and the newer (unpublished) 2011 competition version, which we respectively call LAMA08 and LAMA11. Both planners were run otherwise with default settings, but tuned to find plans faster rather than to find good plans, as advised by LAMA's authors. Some issues with the front-end of LAMA08 were fixed by replacing the front-end with a newer version from the Fast Downward system, following Malte Helmert's instructions. LAMA08 incorrectly reports 6 AIRPORT/ADL instances unsolvable.

These planners are the winners of the non-optimal classical planning tracks of the 2000, 2002, 2008, and 2011 competitions (FF, LPG-td, LAMA08, LAMA11), and a runner-up from the 2004 competition (YAHSP). We did not include winning planners from the 2004 and 2006 competitions. LAMA08 is a successor of the 2004 winner FD and it is implemented in the same general framework. The 2006 winner SGPlan uses specialized solution methods for several of the standard benchmark domains. Overall, the planners we use are frequently used in experimental comparisons of planning algorithms.

6.3. Test Equipment and Setting

All the experiments were run in workstations with Intel Core i7 3930K CPUs running at 4.2 GHz with 32 GB of main memory and a 12 MB L3 cache, under Linux Mint. All planners were run in a single core with the other five cores busy, so that access to the shared L3 cache was not exclusive during the runs. We had the binaries of YAHSP and LPG-td only for the 32-bit x86 instruction set, which meant that these planners could use at most 4 GB of memory. However, these planners terminated because of

a memory overflow only in very few cases. We observed with the other planners for which we could use both 32-bit and 64-bit versions that the 32-bit version is often 20 per cent faster, most likely because of smaller memory use, due to the use of 4-byte instead of 8-byte pointers. We compiled M, Mp, HSP, FF and LAMA from the source files to use the full amd64 instruction set that allows addressing memory past the 4 GB bound. The memory limitations were relevant only for some of the planning competition benchmarks that are experimented with in Section 6.6. With them, LAMA used more than 4 GB of memory for 61 instances, including 20 with over 8 GB. LAMA solved 7 of the instances requiring over 4 GB of memory, and 6 of the ones requiring over 8 GB. For our planners M and Mp, the consideration of long horizon lengths for instances with very high numbers of actions (hundreds of thousands and more) meant allocating large amounts of memory, and we stopped considering longer horizons as soon as 10 GB of memory had been allocated. This affected about a dozen of problem instances. Of instances solved by Mp in 30 minutes, over 4 GB of memory was used for 71 and over 8 GB for 23.

For most of the experiments we used a 1800 second time limit per instance. The 1800 second time limit was chosen to get a good understanding of the relative behavior of the planners on a relatively long time horizon, but also to allow the experiments to be carried out in a reasonable amount of time. As we will see later, all of the planners solved few instances between the 10 and 30 minute marks, and performance differences that showed up past the 3 minute mark are of minor importance. This suggests that the 30 minute time limit is more than sufficient to differentiate between the planners. The reported runtimes include all standard phases of a planner, starting from parsing the PDDL description of the benchmark and ending in outputting a plan.

We ran each planner with each problem instance once. The randomization from Section 4.3 affects the runtimes of our planner on different runs, but not much: different complete runs of all instances solved two instances more or less, depending on whether the runtimes were slightly below or slightly above the time limit. Of the other planners, we ran LPG-td with the random seed 12345 and did not try other seeds. The rest of the planners don't use randomization or use pseudorandom generators with a fixed seed number.

6.4. Confirmation of the Efficiency of Our SAT Solver Implementation

To show that the quality of the implementation of our SAT solver and the VSIDS heuristic matches the state-of-the-art in SAT solving, we compare our SAT solver to the winners of the application/industrial track in the 2007 and 2011 SAT competitions, RSAT [45][4] and Lingeling. The runtimes for solving the planning competition problems (see Section 6.6) are given in Figure 6. We translated the test problems into DIMACS CNF for horizon lengths $0, 5, 10, 15, 20, \ldots, 130$ [5] and solved them with a 180 second time limit per instance[6], and then calculated the corresponding Algorithm B runtimes with $\gamma = 0.9$. The Lingeling and RSAT runtimes exclude the construction of the CNF formulas by our planner's front-end, and for this reason the curves are not completely accurate. The times also exclude the writing and reading of DIMACS CNF files,

With these problem instances, our SAT solver with VSIDS as the decision heuristic outperforms RSAT with all timeout limits until 1800 seconds, and it outperforms Lingeling for timeout limits until about 500 seconds. With timeout limits past 500 seconds Lingeling solves as many instances as M. A main factor in the runtime differences is preprocessing: for many large SAT instances that our planners solve almost instantaneously, RSAT and Lingeling spend considerable time with the preprocessing before starting the search phase. Otherwise the efficiency of our CDCL implementation is almost at the same level as Lingeling and RSAT, in terms of the number of decisions and conflicts per second. Because our SAT solver does no preprocessing and Lingeling strongly relies on it, these solvers' behaviors substantially differ. Lingeling's runtimes are often much higher than M's, up to two or three orders of magnitude, but in some cases the preprocessing pays off and Lingeling scales up better, solving more instances.

[4] We used RSAT 3.0 from 2008, without the SATeLite preprocessor.

[5] For the blocks world problems we used horizon lengths up to 200.

[6] This is sufficient to determine the planners' runtimes up to 1800 seconds with $\gamma = 0.9$.

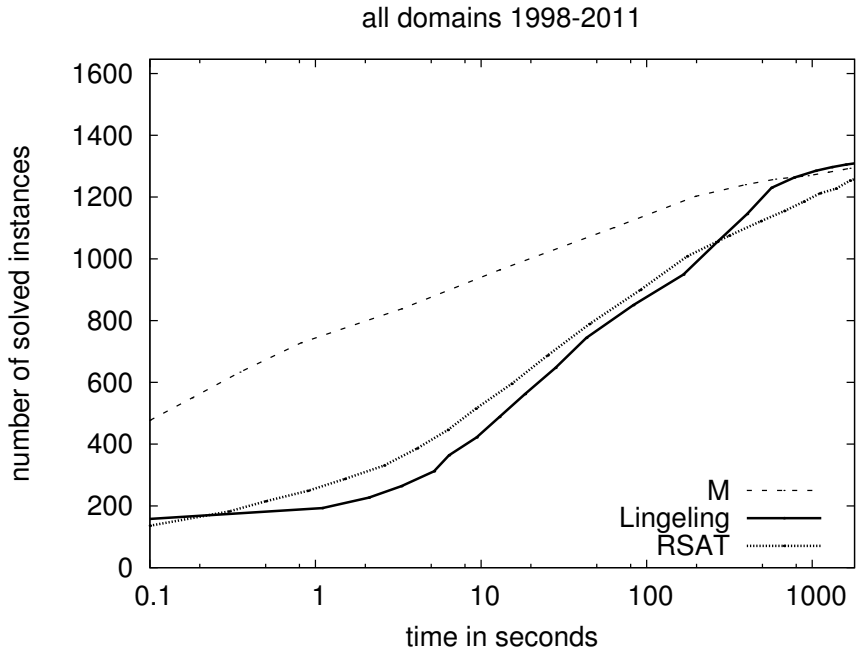


Figure 6: Number of instances solved by different SAT solvers

A peculiarity of SAT problems obtained by translation from the standard planning benchmark problems from the planning competitions, in contrast to SAT problems representing many other applications, is their extremely large size and the fact that these problems can still often be solved quickly. The largest SAT problems Lingeling solves (within the time bounds explained earlier) are instance 41 of AIRPORT (417476 propositional variables, 92.9 million clauses) and instance 26 of TRUCKS (926857 propositional variables, 11.3 million clauses).

Our planner solves instance 49 of AIRPORT (13840 actions and 14770 state variables) with a completed unsatisfiability test for horizon 65, with 1.12 million propositional variables and 108.23 million clauses, and a plan for horizon 85, with 1.46 million propositional variables and 141.54 million clauses. The planner also solves instance 33 of SATELLITE (989250 actions and 5185 state variables), with a plan found for horizon 20, with 19.89 million propositional variables and 69.99 million clauses, backtrack-free in 14.50 seconds excluding translation into SAT and including search effort for shorter horizons. These are extreme cases. More typical SAT instances have less than 2 million propositional variables and a couple of million clauses.

As we will see in Sections 6.5.2 and 6.5.3, all existing planners have difficulties solving much smaller problems that have a more complex structure, with only some dozens of actions and state variables. For these problems, the SAT instances have some thousands of clauses and propositional variables.

6.5. Comparison of Planners with Combinatorially Hard Problems

Different approaches to planning and state-space search have different strengths. When the state space can be easily completely enumerated and the number of states is at most some millions, blind explicit state-space search is generally the strongest approach. The SAT approach is more sensitive to the length of the plans, and cannot always take advantage of the small cardinality of the state space. One strength of SAT has been in solving hard combinatorial planning problems, with state spaces beyond the reach of explicit state-space search, and with a structure too complex to be captured by existing heuristics.

In this section we evaluate the impact of our new heuristics on the solution of such planning problems. We consider translations of hard combinatorial search problems into planning as proposed by Porco et

domain	instances	Mp	M	LAMA08	FF	HSP	YAHSP
clique	600	61	193	37	42	23	61
coloring	280	64	66	1	4	5	18
Hamiltonian	200	82	71	65	52	38	54
k-colorability	480	122	111	114	54	54	113
matching	160	69	80	35	0	0	33
SAT	200	41	78	39	1	20	0
total	1920	439	599	291	153	140	279

Table 2: Number of Porco et al. [46] instances solved by different planners in 300 seconds

al. [46], as well as hard instances generated according to problem parameters that have been empirically determined to be hard for existing algorithms [54].

6.5.1. Graph Problems

Some of the hardest planning problems are those that include hard combinatorial problems as subproblems. Although many hard combinatorial problems appear implicitly or explicitly as subproblems in many planning problems of practical relevance, it is also interesting to study these problems in isolation. Porco et al. [46] have presented a method for generating translations of NP-complete problems to planning, and demonstrated it with several graph problems, including Clique, 3-Coloring, Hamiltonian Circuit, k-Colorability, Matching and SAT. According to Porco et al. [46], our planner M with a general-purpose SAT solver as the search method is the strongest with these problems, but they do not quantify this statement further.

We ran the planners with a 300 second time limit, and summarize the results in Table 2.[7] Unlike Porco et al., whose experiments had most of the instances solved by M in 1800 seconds, we did not use the known plan length lower and upper bounds and could not therefore distinguish "no" answers from timeouts, leading to M solving less than half of the instances, and only ones with a "yes" answer. Similarly to the planning problems that involve solving hard unsatisfiable formulas (see Section 6.6.2), VSIDS is stronger than our new heuristics. For some problems the differences seem minor, but we suspect that major differences would be apparent with instances with negative answers (testing of unsatisfiability), as suggested by the results of Section 6.6.2.

Both SAT-based planners perform considerably better than LAMA, FF, HSP and YAHSP. As half of the problems contained negative preconditions and other features not handled by HSP or YAHSP, we eliminated the unsupported features before running HSP and YAHSP. We do not include data for LPG-td because all instances it solves are due to its second phase which is borrowed from FF.

6.5.2. Solubility Phase Transition

The discovery of the relation between computational difficulty and phase transitions, the relatively abrupt transition from solvable to unsolvable problem instances as a parameter is changed [12, 40], was one of the great advances in understanding the difficulty and structure of hard combinatorial problems such as SAT. The hardest instances of a problem are typically in the parameter range that covers the phase transition region, and instances outside the region are typically easy. Essentially, the phase transition region divides the problem instances to the easy under-constrained, the hard critically constrained, and the easy over-constrained, Phase transitions exist in all kinds of constraint-satisfaction, resource-allocation and planning problems, but in their purest form they have been investigated in the form of models of sampling from the space of all problem instances.

In planning, phase transitions and easy-hard-easy patterns were first observed and investigated by Bylander [11]. The under-constrained instances are those with lots of actions and consequently also lots

[7] We give the results for LAMA08, which performed substantially better than LAMA11.

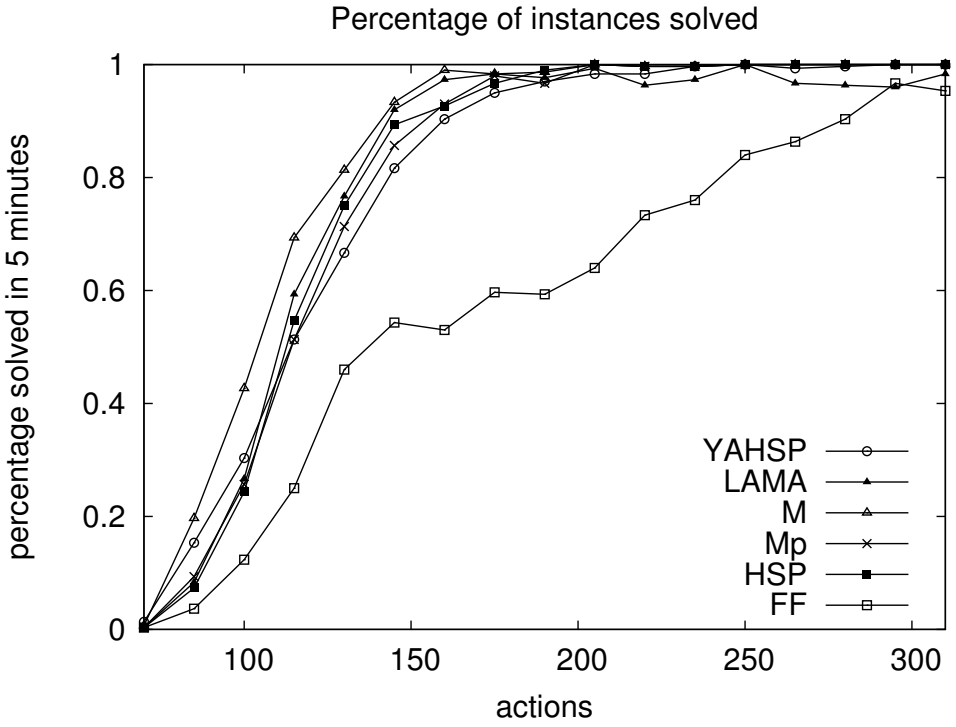


Figure 7: Percentage of instances solved in the phase transition region

of alternative plans (which Bylander showed to be easily solvable by a simple hill-climbing search), the over-constrained are those with few actions and no plans (which Bylander showed to have no plans by a simple syntactic test), and the critically constrained those in between, with a small number of plans which are difficult to find. The parameter values corresponding to the critically constrained problems can be understood in terms of the graph density in random graph models, defined as the ratio of arcs to nodes, and the emergence of the giant component [7] as the density is increased [54]. The critically constrained instances in Bylander's and related models are significantly harder than the planning competition benchmarks (Section 6.6) of the same size: some instances with only 20 state variables are hard, and many with only 40 state variables are very hard [54].

In our experiments, we used model A [54] to generate 4500 instances that covers the easy-hard-easy transition in the phase transition region. The instances have 40 state variables, the actions have 3 preconditions and 2 effects, and there is only one goal state. We test different actions-to-variables ratios from 1.75 to 7.75, corresponding to 70 to 310 actions with a step of 15 actions. For each ratio we generate 300 instances.

The results are shown in Figure 7. As these problem instances are already quite difficult, we don't know which of the hardest instances have plans. Hence our comparison concerns the relative performance of the planners, how many of the instances for each actions-to-variables ratio each planner can solve, and what the median runtimes are. Most of the instances below ratio 2.0 (corresponding to 80 actions) are trivially found unsolvable, as there are not enough actions to reach the goals, and this is detected either by a simple syntactic analysis or with a small amount of search. Practically all of the instances above ratio 4.0 (corresponding to 160 actions) are trivially solvable: there are lots of alternative plans, and almost any search that makes progress towards the goals will almost immediately reach them. The hardest instances are between these ratios. M performs most solidly. Mp, LAMA and HSP solve a moderate percentage fewer instances than M, but there is no big difference between them, and the percentages of solved instances goes up to 100 when the number of actions reaches about 160. The most visibly divergent behavior is by FF, which does not solve many of the very easy instances in 5 minutes. The reason for

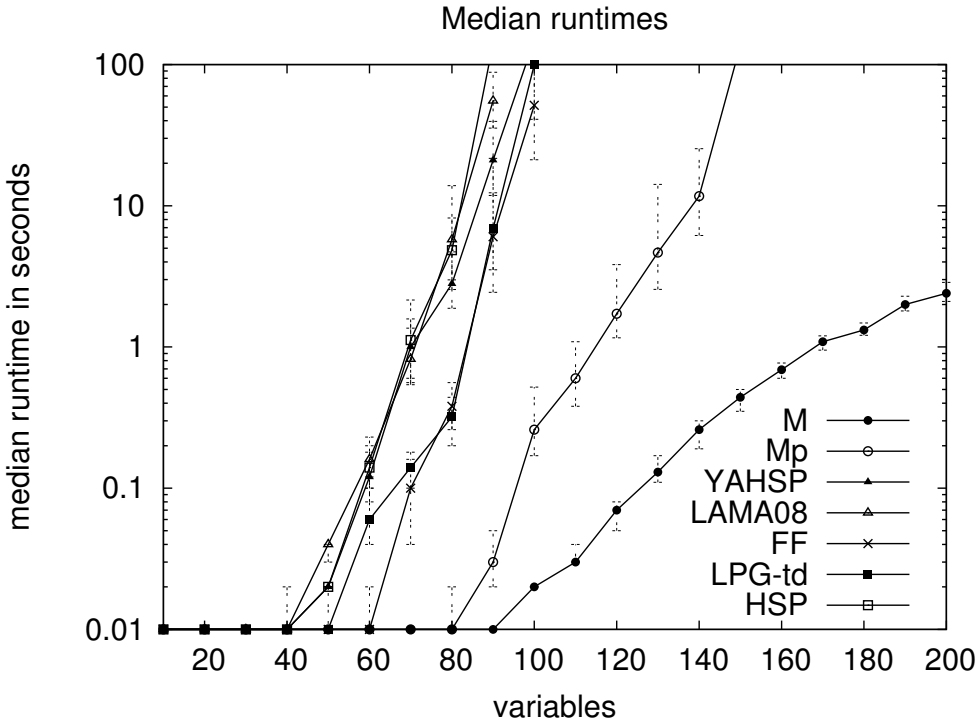


Figure 8: Scalability with problems representing action sequencing

this is the hill-climbing search in FF's first phase, in which pruning with "helpful" actions eliminates too many actions to be able to reach the goals, but which also does not give up and let the HSP-style second phase continue because the local minima are too large to search through exhaustively [50]. YAHSP and LAMA, which also use helpful actions, have a substantially better performance due to their use of helpful actions only for tie-breaking without categorically eliminating all non-helpful from consideration.

6.5.3. Action Sequencing

Another class of hard planning problems, similar in flavor to the phase transition problems above, varies the density of the state transition graph without disconnecting the initial node from the goal node [60]. In these problems, the solubility of an instance is guaranteed by construction: first generate an execution (a state sequence) with a fixed number of state variables changing between two consecutive states, and then for each pair of consecutive states generate an action that is executable in the first state and that modifies it to obtain the second state. The selection of the preconditions determines the difficulty of the instances in terms of the density of the graph and the flexibility in which the actions can be ordered.

We experimented with instances that have N state variables, 2 state variables changing their value between two consecutive states in the sample execution, $\frac{N}{2}$ actions with 3 effects and 2 preconditions in each (with one of the effects having no effect in the sample execution), N goal literals which determine the goal state uniquely, and the parameter $\pi = 4$ determining the difficulty level. We chose π close to what appears to be the most difficult region of instances with a given N [60]. Figure 8 gives the median runtimes as N is increased from 10 to 400, together with 95 per cent confidence intervals, and Figure 9 gives the percentages of solved instances. The new heuristic fares worse than the VSIDS heuristic, but better than LPG-td and the planners that use explicit state space search. The curve depicting the performance of Mp seems slightly less steep than those of HSP, LPG-td, FF, LAMA and YAHSP, and the performance difference between 80 and 100 variables is two orders of magnitude and appears to be increasing. The other planners all have roughly the same performance. Except for some of the smallest

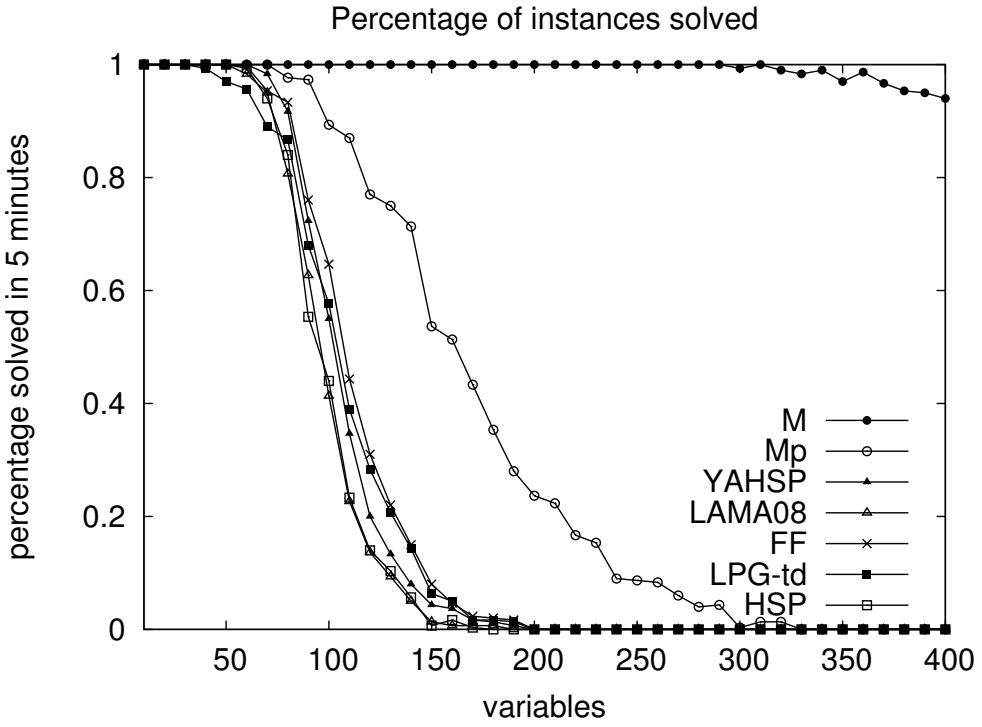


Figure 9: Scalability with problems representing action sequencing

instances, FF always immediately switches from its local search algorithm to the systematic HSP-style search. Similarly, the additional techniques YAHSP and LAMA employ don't substantially help with these problems, and also their performance is close to HSP's. The gap between M and Mp is first similar to the gap between Mp and the rest of the planners, about two orders of magnitude in terms of median runtimes, but then grows quickly as runtimes of M grow significantly slower.

6.6. Comparison of Planners with Competition Benchmarks

Much of the experimentation with algorithms for the classical planning problem has focused on the benchmark problems used in the planning competitions. Next we present the results for these problems.

We included almost all domains used in the competitions from 1998 until 2011 in our experimental comparison, as well as most instances. The excluded domains and others handled exceptionally are the following.

- If a domain was used in more than one competition, we used the instances from the competition that had the harder instances. If the instances were different but there was no difference in hardness, we used the newer instances. The domains used in multiple competitions are listed in Table 3.

- We excluded the 2000 SCHEDULE domain from the comparison because of grounding issues, the overall simple structure of the domain, and the very high number of instances (500). First, none of the planners except LAMA ground the domain quickly. Second, after grounding by LAMA's front-end, Mp and LAMA solve every one of the 500 instances in seconds, but only after spending up to several minutes with preprocessing. The planner that solved almost all of the grounded instances in seconds including preprocessing is YAHSP. Also LPG-td's first phase did very well and would have solved almost all of the series quite quickly had it not had an internal limit of 10000 ground actions.

- Some of the planners were very slow to ground the 1998 LOGISTICS instances, so we grounded them with FF's front-end before running the planners. FF did not parse files with more than about

excluded instances	covered by	justification
2000 LOGISTICS	1998 LOGISTICS	harder
2002 ROVERS	2006 ROVERS	includes the old
2002 FREECELL	2000 FREECELL	harder
2002 SATELLITE	2004 SATELLITE	harder
2006 OPENSTACKS	2011 OPENSTACKS	harder
2006 PIPES/TANKAGE	2004 PIPES/TANKAGE	exactly the same
2008 ELEVATOR	2011 ELEVATOR	harder
2008 OPENSTACKS	2011 OPENSTACKS	harder
2008 PARC	2011 PARC	equally hard
2008 PEGSOL	2011 PEGSOL	equally hard
2008 SCANALYZER	2011 SCANALYZER	equally hard
2008 SOKOBAN	2011 SOKOBAN	harder
2008 WOODWORKING	2011 WOODWORKING	harder

Table 3: Instances excluded from the comparison

100000 ground actions due to a parser restriction, and LAMA was slow to compute invariants from the grounded representation. Since both of these planners could ground the instances quickly, we used the original ungrounded input with them.

- Of the STRIPS domains, 2011 TIDYBOT contains negative preconditions, which are not supported by HSP. We ran HSP with a modified version of TIDYBOT with positive preconditions only, by using the standard reduction which introduces a state variable $\hat{x}$ for every state variable x that occurs negatively in a precondition and forces $\hat{x}$ and x to have opposite truth-values.
- For the 1998 MPRIME domain, we use a corrected version that adds a missing equality test in the *drink* action.
- 1998 ASSEMBLY/ADL uses a syntactic feature in the schematic actions that is not implemented by all of the planner front-ends. We grounded all instances of this domain before running the planners.
- The implementation of equality in YAHSP does not conform to the PDDL definition. To run YAHSP correctly, we added the :equality keyword to the PDDL requirements list in the 2006 TPP and the 2011 SCANALYZER domain files. This forced YAHSP instantiate schema variables with all object combinations, including ones in which more than variable are instantiated with the same object.

6.6.1. Comparison of Different Configurations of Our Planner

The impact of the heuristics from Section 4 on the performance of the variable selection scheme is illustrated in Table 4. The baseline `---` planner solves 1188 of the 1646 planning competition instances from Section 6.6 in 30 minutes. For other configurations (as described in Table 1), we show the improvement obtained as the difference between the number of solved instances and 1188. The goal ordering heuristic from Section 4.1 and the action choice heuristic from Section 4.2 lead to a minor improvement over a fixed goal ordering and an arbitrary selection of actions. The replacement of strict backward chaining depth-first search with the less directional form of search from Section 4.3 is a substantial improvement, without which the performance would be slightly below the level of VSIDS.

6.6.2. Comparison to VSIDS with a Focus on Unsatisfiable Formulae

We compare the new heuristic to the VSIDS heuristic. Almost all of the currently strongest implementations of the CDCL algorithm use some variant of VSIDS or a related heuristic.

To compare heuristics in terms of both satisfiable and unsatisfiable formulas, with emphasis on the unsatisfiable ones that are required for proving the minimality of the horizon length, we set up our planners

	??-	??m	??w
--?	1188	1376 (+188)	1383 (+195)
a-?	1201 (+13)	1385 (+197)	1399 (+211)
-g?	1213 (+25)	1374 (+186)	1399 (+211)
ag?	1225 (+37)	1404 (+216)	1416 (+228)

Table 4: Impact of different features on the planner's performance

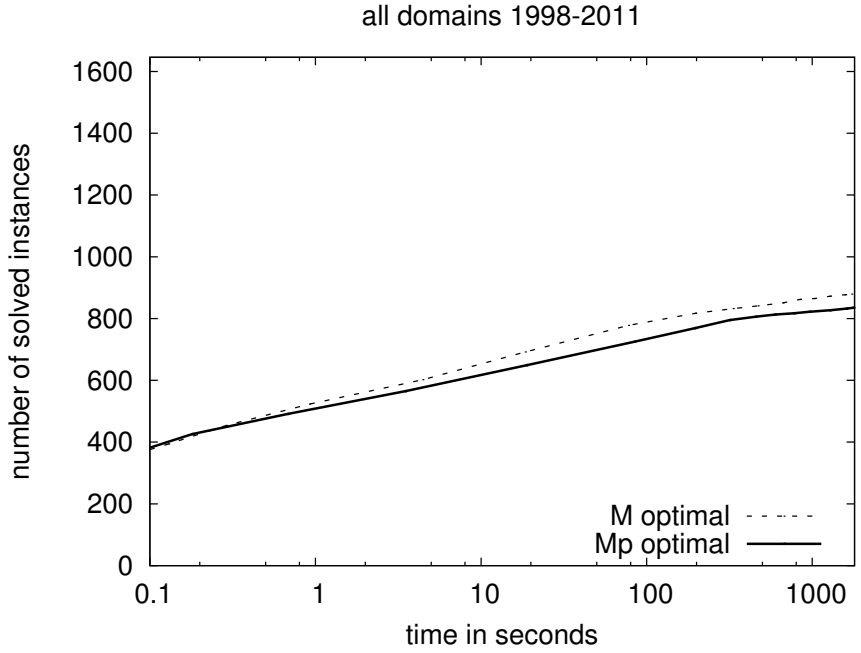


Figure 10: Number of instances solved in a given time with the sequential strategy

to use the BLACKBOX sequential strategy which goes through horizon lengths 0, 1, 2, 3 and so on, until it finds a satisfiable formula. For many problem classes, including the planning competition problems, the runtimes of the planners in this configuration are very strongly dominated by the unsatisfiable formulae. The results for these problems are summarized in Fig. 10. The plot shows the number of problem instances that are solved (finding a plan) in n seconds or less when using VSIDS and when using the new heuristic. The solver with VSIDS solves about 10 per cent more instances when 1800 seconds is spent solving each problem instance. With the sequential strategy, usually almost all of the computation effort is spent solving the last unsatisfiable formulas right before the first satisfiable one. However, as we will see later, VSIDS is weaker than the new heuristic with satisfiable formulas, which are far more important when finding plans.

6.6.3. Comparison to VSIDS in Terms of Plan Sizes and Runtimes

In Figures 11 and 12 we compare the solution times and plan sizes for VSIDS and the new heuristic, as implemented in our planners M and Mp respectively, with parallel solution strategies that don't require completing the SAT solving for unsatisfiable formulae. Each dot in these figures represents one problem instance, and the location of the dot on the X-axis depicts the runtime or the plan size with our planner with the VSIDS heuristic, and its location on the Y-axis that of our planner with the new heuristic. Hence any dot on the diagonal means that the planners perform equally well, and dots below and right mean that the runtime or the plan size is higher with VSIDS than with the new heuristic.

Figure 11 shows that for a vast majority of the problem instances the new heuristic outperforms

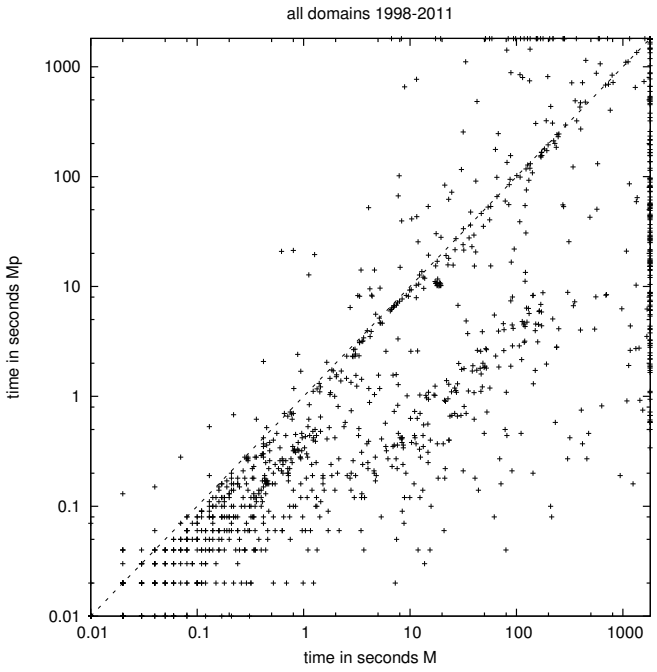


Figure 11: Runtimes with the new heuristic (Mp) and with VSIDS (M)

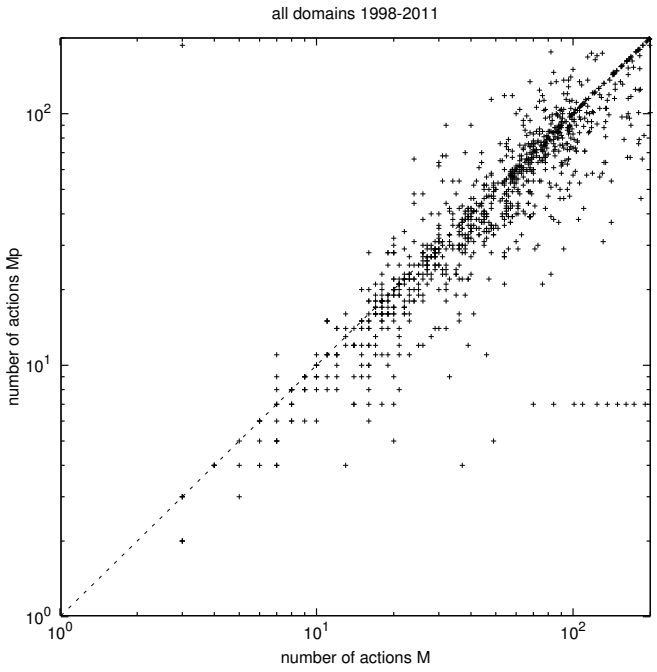


Figure 12: Plan sizes with the new heuristic (Mp) and with VSIDS (M)

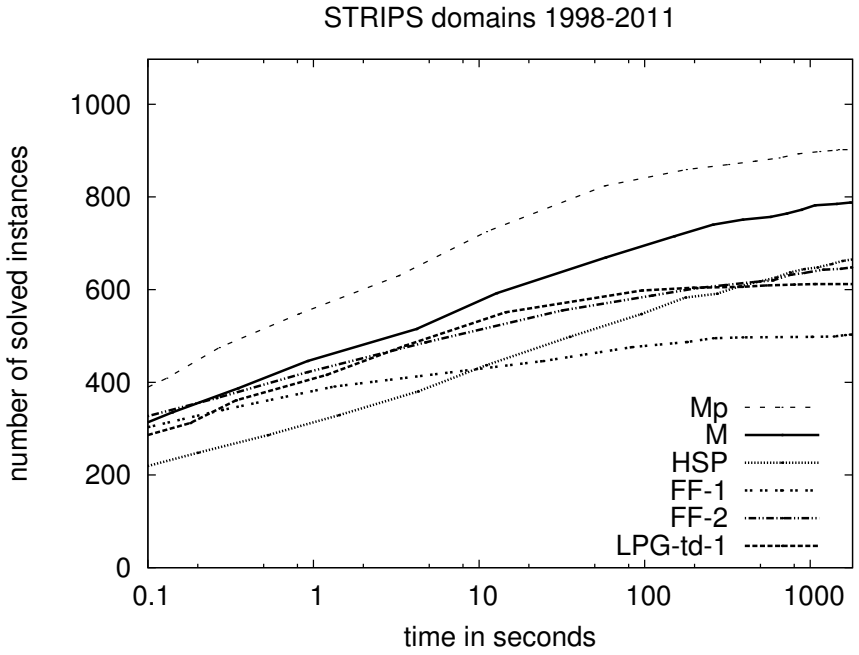


Figure 13: Number of STRIPS instances solved by different algorithms

VSIDS, often by 1 or 2 orders of magnitude. VSIDS is sometimes faster, but only in about two dozen cases more than by a factor of 10. There is overall a high variation in the runtimes of the CDCL algorithm for a given instance due to the arbitrariness of some of the branching decisions, and for this reason one would in general see a weaker algorithm outperform an overall stronger one in a small number of cases, exactly as we have observed here. Plans with VSIDS are on average a bit longer than with the new heuristic, as indicated by Figure 12, but the differences are relatively small. The longer plans are mostly due to redundant actions that don't contribute to any of the goals or preconditions in the plan, and which could be eliminated by a simple post-processing step.

6.6.4. Comparison to Other Search Algorithms

We first compare our planners to what could be viewed as the baseline search algorithms in the different approaches, including the planners and planner components that use the standard best-first search algorithm with a heuristic but no additional pruning, shortcut or preference mechanisms. These are HSP and the component algorithms of FF and LPG-td. Then we follow with the rest of the planners, including FF and LPG-td themselves as well as LAMA and YAHSP that enhance the baseline HSP-style search with additional techniques. Finally, we have a look at the impact of our planners in the big picture of planning by considering algorithm portfolios that can be built from the individual planners.

The planners or planner components we compare to and that are based on only one search algorithm and one heuristic are HSP, the two phases of FF [26], and the first phase of LPG-td [18]. The runtimes for the first phase of FF are without the goal agenda mechanism, as this mechanism is orthogonal to the other features of the planner and it could be equally used in any other planner. The goal agenda increases the number of instances solved in 1800 seconds by 77, being critical for Blocks World but having no impact for most other domains.

Figure 13 illustrates the performance of these planners or planner components. All are outperformed by our baseline SAT-based planner M from 2006 [50]. A remarkable fact is that M has an outstanding performance although it uses a generic SAT heuristic which is completely unaware of planning. Explicit state-space search similarly without a heuristic would perform extremely poorly with these problems because there are far too many states to go through exhaustively.

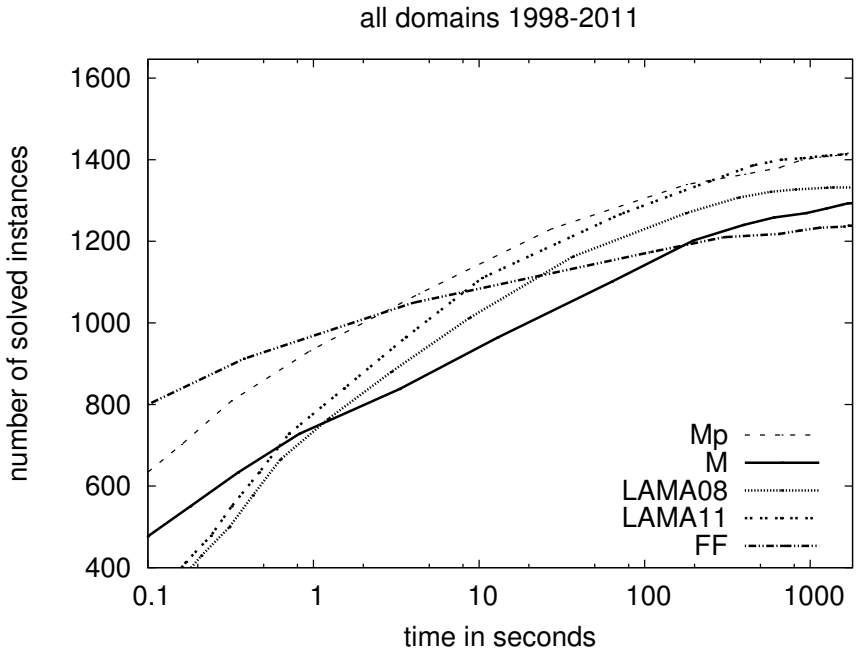


Figure 14: Number of instances solved by different planners

6.6.5. Comparison to Other Planners

Next we make a comparison between our planners and FF and LPG-td, which consist of the components evaluated in the previous section, as well as LAMA08, LAMA11 and YAHSP, which consist of one phase only, but employ additional techniques on top of the basic heuristic search algorithm. A diagram depicting the performance of M, Mp, LAMA and FF with all instances is given in Figure 14. For the subclass of STRIPS instances, and including also HSP, YAHSP and LPG-td which only support STRIPS, a diagram is given in Figure 15.

The curves in all cases are similar: all planners solve a large fraction of the problem instances in seconds, and the number of solved instances increases slowly as the time limit is increased past a couple of minutes. In Table 5 we break down a part of the results to different benchmark domains, showing the numbers of problem instances solved in 1800 seconds. We also calculate a score, as the sum of the percentages of instances solved for each domain in 30 minutes, and estimate the statistical significance of the scores by calculating confidence intervals. The confidence intervals are calculated by a bootstrapping procedure, hypothesizing that the planning competition domains (and instances) are randomly sampled from some larger pool of similar domains. For Mp we give the 95 per cent confidence interval upper and lower bounds, and for the other planners we calculate the intervals as the difference to the Mp score as obtained with the bootstrap calculation. According to this calculation, the difference between Mp and M is "significant", and the difference to FF is also "significant", but only with a small margin. Differences between Mp and LAMA08/LAMA11 are not "significant" according to this calculation: when drawing samples of domains from the hypothetical domain pool, Mp would often get a score that is higher than that of LAMA11, and LAMA11 would often get a higher score than Mp.

Our new heuristic is an improvement over VSIDS in almost all domains. With many of the easiest domains and instances the improvement is however modest, as there is not much room to improve and the runtimes are often dominated by the preprocessing phase shared by the planners.

There are only four domains where the new heuristic is not an improvement over VSIDS. With many of the instances of OPTICAL-TELEGRAPH and TRUCKS (both STRIPS and ADL), the new heuristic is more effective than VSIDS, but for a number of instances VSIDS finds a plan within the 30 minute time bound while the new heuristic does not. With BLOCKSWORLD the VSIDS heuristic scales up clearly

		LAMA				
		Mp	M	2008	2011	FF
1998-GRID	5	5	3	5	5	5
1998-GRIPPER	20	20	20	20	20	20
1998-LOGISTICS	30	30	30	29	30	30
1998-MOVIE	30	30	30	30	30	30
1998-MPRIME	20	20	18	20	20	19
1998-MYSTERY	19	19	18	19	14	16
2000-BLOCKS	102	63	82	54	95	80
2000-FREECELL	60	45	32	59	59	60
2002-DEPOTS	22	22	22	18	22	22
2002-DRIVERLOG	20	20	19	20	20	16
2002-ZENO	20	20	18	19	20	20
2004-AIRPORT	50	50	48	38	38	39
2004-OPTICAL-TELEGRAPH	14	14	14	3	14	13
2004-PHILOSOPHERS	29	29	29	12	14	14
2004-PIPESWORLD-TANKAGE	50	38	11	38	41	22
2004-PIPESWORLD-NOTANKAGE	50	41	20	44	44	36
2004-PSR-SMALL	50	50	50	50	50	43
2004-SATELLITE	36	35	35	31	36	36
2006-PATHWAYS	30	30	30	28	28	20
2006-ROVERS	40	40	40	40	40	40
2006-STORAGE	30	30	25	21	20	18
2006-TPP	30	30	30	30	30	28
2006-TRUCKS	30	21	22	8	15	11
2008-CYBER-SECURITY	30	30	30	29	29	4
2011-BARMAN	20	10	0	17	20	0
2011-ELEVATORS	20	20	1	20	20	20
2011-FLOORTILE	20	20	20	2	6	5
2011-NOMYSTERY	20	17	17	13	18	4
2011-OPENSTACKS	20	0	0	18	20	20
2011-PARCPRINTER	20	20	20	12	20	20
2011-PARKING	20	0	0	20	20	8
2011-PEGSOL	20	20	19	19	20	20
2011-SCANALYZER	20	20	13	20	20	20
2011-SOKOBAN	20	2	0	13	19	17
2011-TIDYBOT	20	17	2	14	16	15
2011-TRANSPORT	20	4	0	16	19	9
2011-VISITALL	20	0	0	20	7	4
2011-WOODWORKING	20	20	20	16	20	4
1998-ASSEMBLY-ADL	24	24	23	24	23	24
2000-ELEVATOR-SIMPLE	150	150	150	149	150	150
2000-SCHEDULE-ADL	150	150	150	134	138	134
2002-SATELLITE-ADL	20	20	20	20	20	20
2004-AIRPORT-ADL	50	49	47	31	45	30
2004-OPTICAL-TELEGRAPH-ADL	48	39	41	19	1	17
2004-PHILOSOPHERS-ADL	48	48	48	23	14	14
2006-TRUCKS-ADL	29	16	22	17	14	11
2008-OPENSTACKS-ADL	30	18	15	30	30	30
total	1646	1416	1304	1332	1414	1238
weighted score	47	39.06	34.36	37.97	40.48	34.11
confidence interval low		34.82	-7.86	-6.09	-3.33	-9.69
confidence interval high		42.79	-1.98	4.14	6.36	-0.12

Table 5: Number of problems solved in 1800 seconds by domain

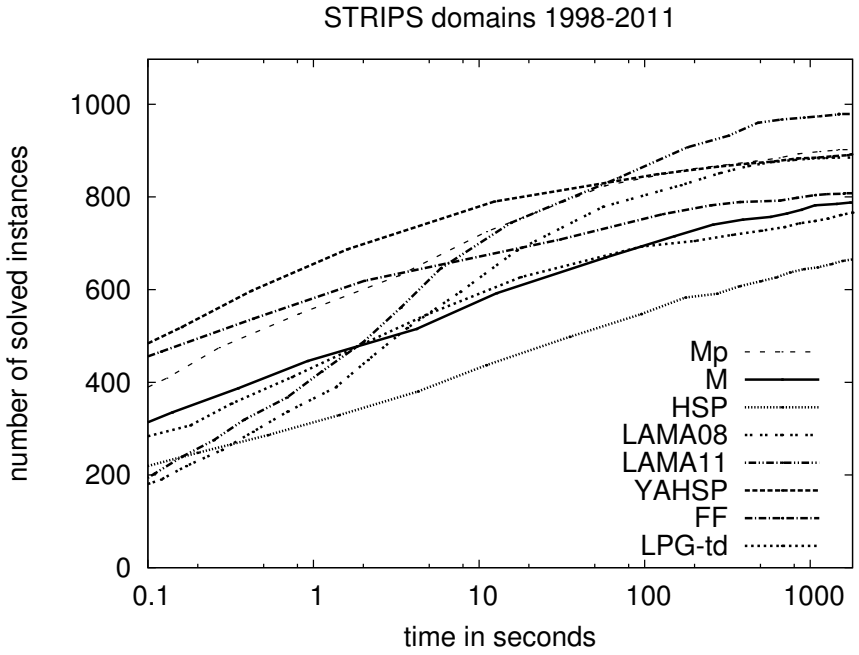


Figure 15: Number of STRIPS instances solved by different planners

better due to its ability to quickly shift to long horizon lengths by completing unsatisfiability tests faster. With CYBERSECURITY, VSIDS is often 15 to 50 per cent faster, and equally often slower. However, both planners solve all instances of CYBERSECURITY in well under one minute.

Overall, the number of cases in which VSIDS is stronger is much smaller than of those where the opposite holds: for several domains the new heuristic dramatically outperforms VSIDS, and for most of the rest the runtimes are a clear improvement over VSIDS. As we have seen earlier, with a number of other types of planning problems than the ones from the planning competitions, especially smaller and combinatorially harder ones, CDCL with VSIDS continues to be the strongest search method, and the improvements over VSIDS are with the type of problems used in the planning competitions.

The new planner often compares well with LAMA11 [48], the winner of the non-optimal non-temporal track of the 2011 planning competition.[8] Figure 16 illustrates the relative performance of LAMA11 and our planner Mp with all of the problem instances in terms of runtime, and Figure 17 in terms of plan size. Diagrams comparing runtimes and plan sizes for each domain separately are given in the appendix. In dozens of cases, the strengths of LAMA and Mp are quite complementary, one planner outperforming the other by two or more orders of magnitude in runtime. Also, both planners in some cases produce much longer plans than the other planner, Mp more so, but for a vast majority of problem instances the plan sizes are close to each other. Of the planning competition instances that are solved by both Mp and LAMA11, the average length of plans found by Mp is 81.72 and of those found by LAMA11 is 72.93.

Earlier, the strength of SAT-based planning has been perceived to be in small but combinatorially hard planning problems, a perception which is to some extent confirmed by the experiments in Sections 6.5.1, 6.5.2, and 6.5.3. However, with the newest planners and concerning the planning competition benchmarks, this is less clearly the case. Figure 18 depict the ratio of the runtimes of LAMA11 and Mp on all of the planning competition instances solved by both planners, plotted against the numbers of actions in the plans found by LAMA11. Instances that LAMA11 solves faster than Mp are below the line corresponding

[8]Note that the evaluation criterion in the competition was the quality of the plans generated, whereas in our comparison we are only counting the number of instances solved. In the 2011 competition, the problem instances were selected so that most of the participating planners could solve them.

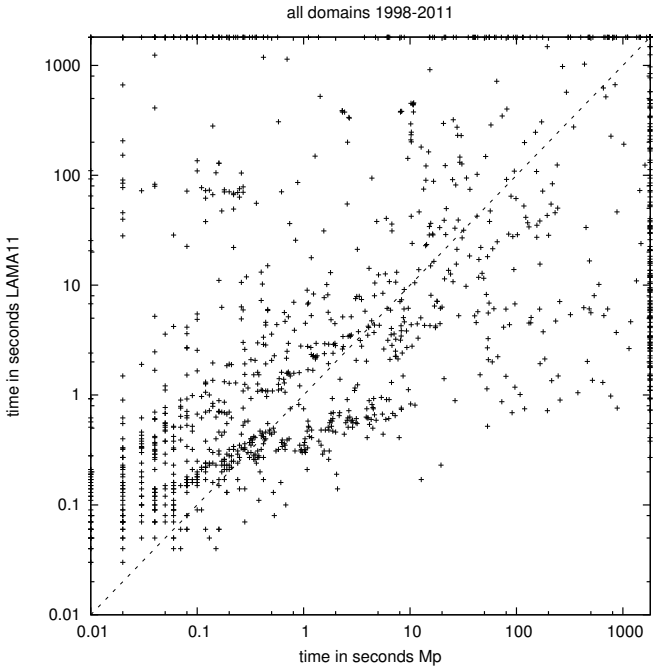


Figure 16: Runtimes with Mp and LAMA for each problem instance

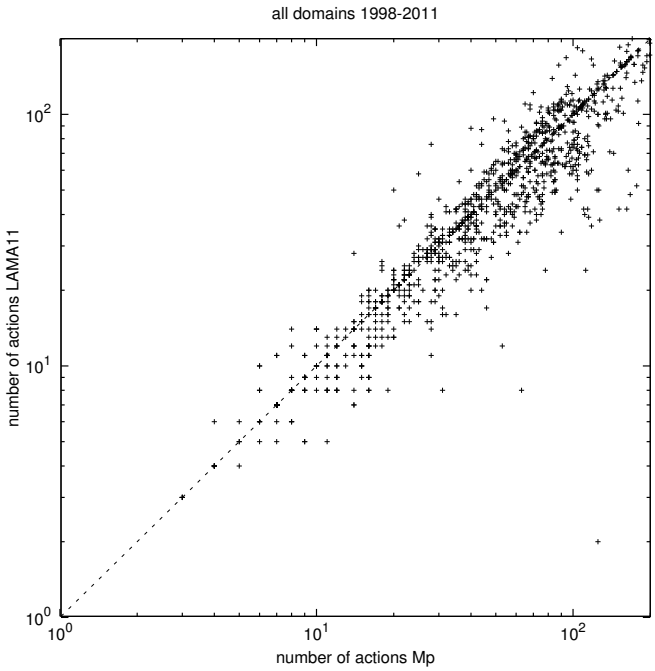


Figure 17: Plan sizes with Mp and LAMA for each problem instance

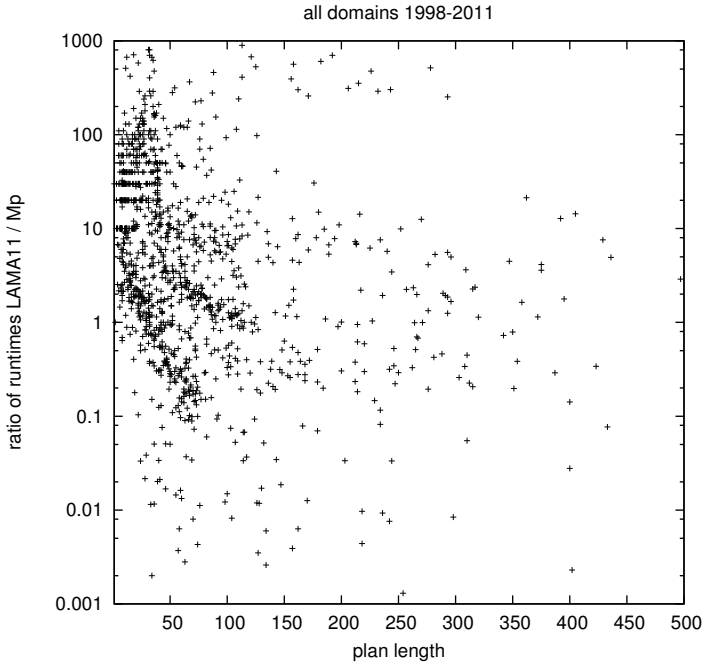


Figure 18: Relative performance of LAMA11 and Mp with increasing plan size

to the X-axis at 1, and instances it solves slower are above the line. The number of instances above the line is roughly equal to the number of dots below the line. Although for some domains LAMA's relative performance improves as the difficulty in terms of the number of action in plans increases, as can be seen from the cloud of dots at the area between $(10, 1)$ and $(50, 0.1)$, this trend is not generally very clear, and there are also several domains for which Mp's relative performance over LAMA11 improves. A plot against the number of state variables or number of ground actions in the instance looks similar. It seems fair to say that with the newest planners, the strengths of SAT-based planners are no longer limited to small and hard instances, but also cover many problems that are large and easy (relative to their size).

For some of the domains there are partial explanations for the performance differences to LAMA11. The formalizations of some of the domains are particularly unfavorable to search directions other than forward search (explicit state space search). A typical issue is incrementing a counter c from i to $i + 1$ where i is in some range $l \leq i \leq u$. This increment can be represented as $u - l - 1$ STRIPS actions with the precondition $c = i$ for $l \leq i < u$ and the effect $c := i + 1$, with each counter value represented as a separate state variable. With forward search this representation is unproblematic as the old value of the counter is always known: only one of the actions, with the precondition matching the current value $c = i$ can be chosen. But with backward search and SAT, for actions in the middle of the plan, selecting an action always necessarily commits to one value of the counter. The problem is that the previous and the next actions in the plan should have compatible values, but at the time of selecting this action it is generally not known what and where these actions will be, often leading to poor action choices that are essentially bad guesses about the values of the counter. Domains with this type of counter increments and decrements are 2000 FREECELL, 2011 BARMAN, 2011 TRANSPORT and 2011 OPENSTACKS, with counters representing container or vehicle capacities and the availability of other resources. With BARMAN, a minor modification of the action description, involving conditional effects, turns the domain from 10 solved to 19 out of 20 solved. With TRANSPORT, the same modification increases the number from 4 to 13. A better representation of the increments than the one used in these modifications at the PDDL level would be possible at the SAT level, leading to substantially smaller SAT instances. More generally, the problem with these four domains is the low abstraction level offered by PDDL/STRIPS, which forces representation decisions at the modeling time which may be good for some search methods

	Mp	M	LAMA08	LAMA11	FF
Mp	1416	1436	1538	**1561**	1507
M	1436	1293	1541	**1556**	1468
LAMA08	1538	**1541**	1332	1471	1461
LAMA11	**1561**	1556	1471	1414	1448
FF	**1507**	1468	1461	1448	1238

Table 6: Number of instances solved in 1800 seconds by 2-planner portfolios

and bad for others. LPG-td, which also does not use explicit state space search (forward search) in its first phase, scales poorly with three of these domains, but not with OPENSTACKS. LPG-td solves OPENSTACKS efficiently due to its ability to increase the horizon length quickly.

Another domain with which our planners perform poorly, and which differs from all other domains, is 2011 VISITALL. The plans in this domain are extremely long, with thousands of actions, with no possibility to parallelize them. Our planner's strategy to consider horizon lengths 0, 5, 10, 15, and so on, the restriction to at most 20 simultaneous horizon lengths, and the difficulty to prove non-trivial lower bounds for horizon lengths, mean that the planner never proceeds further than a couple of hundred plan steps, and never finds any plans. If we force the planner to consider horizon lengths 1000, 2000, and so on, the new heuristic (but not VSIDS) finds plans for the first instances of VISITALL quickly with little or no search. Very long horizons remain problematic to SAT-based planners because of the high memory requirements that follow from the need to represent all actions and state variables for every time point.

There might seem to be a discrepancy between the performance differences of FF and M in Table 5 and in our earlier article [50], with the new results showing that the difference between M and FF is small, whereas the 2006 article seemed to suggest a far bigger difference. One factor in the difference is improvements in implementations of SAT solvers since 2006. In 2006, we used the Siege SAT solver [63], which is dramatically outperformed by newer solvers and our own solver. Second, M considers only every fifth horizon length, 0, 5, 10, 15 and so on, whereas in the 2006 paper we considered all horizon lengths, 0, 1, 2, 3, and so on, and in some cases did not go far enough to discover the easiest satisfiable formulae. And, finally, the data given in the 2006 paper did not include those problems that were very quickly solved by our planner, giving an overly negative impression of its performance.

6.7. Impact of the New Heuristic on Portfolios

There is the obvious question about the performance of our planner as a component of an algorithm portfolio. We consider portfolios that consist of two planners run in parallel, each planner getting 50 per cent of the CPU, and a plan is returned as soon as one of the planners finds one. Other ways of combining planners are possible, including sequential composition with a fixed amount of time allocated to each planner. An advantage of parallel composition is that it is symmetric with respect to the components, so that if one of the components delivers a solution quickly, the parallel portfolio will do it as well.

We have Tables 6, 7, and 8 illustrating the 2-planner portfolios that can be constructed by parallel composition. Table 6 lists portfolios for planners that support the PDDL language with conditional effects. Table 7 lists portfolios for the baseline search algorithms for each approach (SAT, explicit state space search, LPG-td), with performance data restricted to the STRIPS instances, and Table 8 lists all planners and planner components, with data similarly restricted to the STRIPS instances.

For each portfolio, the tables show the number of problem instances solved in 30 minutes. The diagonal represents the planner run alone, getting 100 percent of the CPU for 30 minutes. For each row we highlight the column with the highest number of solved instances.

M and Mp are relatively stronger, and complement the other planners better, when considering the full set of instances, including ones in the general PDDL language with conditional effects. When restricted to STRIPS instances, LAMA11 is generally the best complement, although M and Mp are in several cases close to LAMA11.

	Mp	M	HSP	FF-1	FF-2	LPG-td-1
Mp	902	920	**966**	942	954	941
M	**920**	781	878	872	894	831
HSP	**966**	878	665	788	724	812
FF-1	**942**	872	788	503	752	739
FF-2	**954**	894	724	752	648	819
LPG-TD-1	**941**	831	812	739	819	612

Table 7: Number of STRIPS instances solved in 1800 seconds by 2-algorithm portfolios

	Mp	M	HSP	FF-1	FF-2	LPG-td-1	LAMA08	LAMA11	FF	LPG-td	YAHSP
Mp	902	920	966	942	954	941	1010	**1043**	991	960	1023
M	920	781	878	872	894	831	1007	**1031**	944	878	1015
HSP	966	878	665	788	724	812	935	**999**	847	844	946
FF-1	942	872	788	503	752	739	907	**979**	807	829	912
FF-2	954	894	724	752	648	819	921	**977**	807	820	912
LPG-TD-1	941	831	812	739	819	612	960	**1007**	879	744	980
LAMA08	**1010**	1007	935	907	921	960	885	1003	973	985	982
LAMA11	**1043**	1031	999	979	977	1007	1003	979	988	1008	1037
FF	**991**	944	847	807	807	879	973	988	808	880	969
LPG-TD	960	878	844	829	820	744	985	**1008**	880	766	988
YAHSP	1023	1015	946	912	912	980	982	**1037**	969	988	892

Table 8: Number of STRIPS instances solved in 1800 seconds by 2-planner portfolios

The strongest portfolio is that of Mp and LAMA11, both with the set of all instances and with the STRIPS subset. With STRIPS instances, several other portfolios are very close, including LAMA11-M, LAMA11-YAHSP, Mp-YAHSP, M-YAHSP and LAMA11-LPG-td. Overall, the differences between the planners in terms of the planning competition instances are far smaller than with the other classes of problems in Sections 6.5.1, 6.5.2 and 6.5.3.

7. Related Work

7.1. Planning with SAT and Constraint Satisfaction

All earlier SAT-based planners used generic SAT solvers, with VSIDS and similar heuristics, and with the breadth-first-style sequential solving of SAT instances for different horizon lengths. All the performance differences in earlier planners came from the SAT solver used and from differences in the encodings, primarily the size the encodings and the use of additional constraints to prune the search spaces.

The best-known early planner that used SAT is BLACKBOX by Kautz and Selman [33]. Rintanen et al. [50] demonstrate that their $\forall$-step semantics encoding is often substantially faster than the BLACKBOX encoding, sometimes by a factor of 20 and more. Both encodings use the same definition of parallel plans. Results of Sideris and Dimopoulos [66] indicate that newer planners in the BLACKBOX family implement encodings that are not faster than BLACKBOX's and are sometimes twice as slow, due to weaker unit propagations. Robinson et al. [62] propose a factored encoding of $\forall$-step plans and demonstrate substantial speed-ups over some of the encodings from the BLACKBOX family. Other recent works claim improvements over Kautz and Selman style encodings [47, 27], but only demonstrate moderate improvements and make no comparison to encodings by Rintanen et al. or Robinson et al.

The more relaxed notion of parallel plans used in our planner, the $\exists$-step semantics [50, 70], allows shorter horizons and smaller formulas than $\forall$-step plans, and therefore leads to substantial efficiency

improvements. This and parallelized search strategies [57] often mean further one, two or more orders of magnitudes of speed-up over other SAT-based planners.

7.2. Planning with Partially Ordered Representations: Graphplan, LPG, CPT

The Graphplan algorithm [6] uses backward search constrained by the planning graph structure which represents approximate (upper bound) reachability information. The action selection of GraphPlan's search may resemble our action selection: given a subgoal l at time t, the choice of an action to reach l is restricted to actions in the planning graph at level $t-1$. This same constraint on action selection shows up in any extraction of action sequences from exact distance information, for example in BDD-based planning [13] and related model-checking methods, and the data structures representing the distances (the planning graph or the BDDs) are not used as a heuristic as in our work: when the action choice for achieving l is not restricted by the contents of the planning graph (which is usually the case), Graphplan will choose an arbitrary action with l as an effect. Another major difference is of course that our heuristic leverages on the inferences and learned clauses of the CDCL algorithm. This is the main reason why our heuristic, despite its extreme simplicity, is as effective as substantially more complex heuristics used with explicit state-space search.

The LPG planner [18] does stochastic local search in the space of incomplete plans with parallel actions similar to the SAT-based approach. LPG's choice of actions to be added in the current incomplete plan is based on the impact of the action on violations of constraints describing the solutions. A main difference between LPG and SAT-based planning is that LPG, similarly to local-search algorithms for SAT, does not use general logical inference, but only a restricted form for propagation of values of non-changing facts from a time point to its predecessors or successors.

Vidal and Geffner [68] present the CPT planner which covers both classical and temporal planning. It uses a constraint-based model and can be viewed as an instance of the partial-order causal link (POCL) framework [38]. CPT's partial plans are partial valuations of the variables expressing the times the actions take place. As in the POCL framework, planning proceeds by identifying *flaws* which suggest possible violations of the constraints in the current partial plan, and then posting additional constraints to eliminate the flaw. As in LPG, the heuristics in CPT evaluate the different ways of removing the flaws in terms of the distances between plan elements related to the flaws in question.

7.3. Planning with State Space Search

Systematic algorithms for heuristic search [44] have long been a leading approach to problem solving in AI, but its use in planning (which is problem solving with a generic high-level input language) was limited until Bonet et al. [9, 8] demonstrated the power of these algorithms and automatically derived heuristics in the HSP planner. Research quickly focused on explicit state-space search guided by heuristics derived from declarative problem descriptions in a generic, problem-independent manner.

The HSP family of planners have to evaluate all of the possible successor states in order to choose one which is likely to lead toward the goal states. In contrast, our work has demonstrated that in the CDCL framework, the current partial valuation gives reliable heuristic information about which actions to add to the current partial plan, without evaluating all action candidates separately, simply by reading the next action (decision variable) from the current partial valuation. While our heuristics are simpler, the inferences and learning of the CDCL framework are more complex than the explicit state-space framework, representing a different trade-off in resource use. Interestingly, the number of states evaluated per second by planners like LAMA is typically within one order of magnitude from the number of decisions (action selections) in Mp or generic VSIDS-based CDCL implementations. Table 9 gives, for four problem instances for which both Mp and LAMA had similar and relatively high search times, the numbers of state expansions and action selections per second. LAMA's numbers of states generated, but not evaluated, per second are considerably higher. Of course, because of the fundamentally different problem representations used by Mp and LAMA, and the fact that a decision (action selection) in the CDCL context could be viewed as a lower level operation than a state evaluation in explicit state-space search, these numbers are not directly comparable.

	LAMA11					Mp		
instance	time	eval	per sec	generated	per sec	time	decs	per sec
2004 AIRPORT 46	117.30	54822	467.37	249761	2129.25	113.81	40312	352.20
2004 PIPES 36	88.08	14049	159.50	713299	8098.31	61.51	39963	649.70
2004 SATELLITE 36	150.56	17313	114.99	35689168	237042.83	10.34	16752	1620.12
2006 TRUCKS 12	162.52	1669395	10271.94	37026952	227830.16	63.31	139027	2195.97
2011 ELEVATOR 17	83.02	24254	292.15	2101104	25308.41	69.04	84347	1221.71

Table 9: Rates of state evaluations and generations for LAMA11 and of action selections (decisions) Mp

There is a resemblance between our variable selection scheme and the *best supporters* and *minimal paths* of Lipovetzky and Geffner [35], in both cases directly going back to the preference for *shortest* possible action sequences. Our variable selection scheme chooses one of the earliest possible actions (with respect to the current partial valuation of the CDCL algorithm) that can make a given (sub)goal true, whereas the minimal paths are sequences of actions constructed by backward chaining so that an action supporting the preconditions of a later action are best supporters in the sense that their value according to the h_{max} heuristic is the lowest. Unlike in our work, the restriction to best supporters is a pruning technique, and not a heuristic, and it leads to incompleteness in Lipovetzky and Geffner's framework [35].

7.4. Domain-Specific Heuristics for SAT Solving

Not much is known about using problem specific heuristics in SAT solving or the workings of SAT solvers when solving planning problems. Beame et al. [4] demonstrate the utility of a problem-specific variable selection heuristic for a clause-learning algorithm solving a combinatorial problem (pebbling formulas), leading to improvements in finding resolution refutations with CDCL.

Our decision heuristic focuses on action variables, and only assigns fact variables at the last stages to complete the assignment that is already known to represent a plan. Theoretical results indicate that the efficiency of CDCL is sometimes decreased if variable assignments are restricted to a subset of the variables only, even if those variables are sufficient for determining satisfiability and unsatisfiability [25, 30]. However, these results, and all other known restrictions on SAT solving efficiency (in a given proof system), only apply to unsatisfiability proofs, which are of limited importance when finding a plan without having to prove the optimality of the plan.

8. Conclusions and Future Work

The contribution of this paper is a simple yet powerful variable selection strategy for clause-learning SAT solvers that solve AI planning problems, as well as an empirical demonstration that the strategy outperforms VSIDS for benchmarks from the planning competitions. With smaller but combinatorially harder problems VSIDS continues to be the strongest heuristic. A main additional benefit over VSIDS is that the variable selection strategy is understandable in terms of the planning problem. This makes it particularly promising because the features that make it strong are largely complementary to the important features of VSIDS, suggests ways to combine them. This is a focus of future work.

Our heuristics ignore many aspects of action selection that have traditionally been considered important, especially in early works on planning. One such issue is interference between different subgoals, caused by conflicts between the actions fulfilling them. With some problems with which our planner did not perform very well we observed such interference issues. Handling action selection and subgoal interactions in a more informed fashion is one avenue to still more effective heuristics.

The main ideas in this work are quite general and could be easily adapted to other applications of SAT and constraint-satisfaction to reachability, for example in LTL model-checking [5] and diagnosis [24], as well as to other forms of planning, for example planning with more complex models of time and with continuous state variables (hybrid systems), by using SAT modulo Theories (SMT) solvers [10, 1, 19, 2],

and planning with nondeterministic actions and partial observability by using quantified Boolean formulae [52, 55] or stochastic satisfiability [36].

Acknowledgments

We thank Hector Geffner and Patrik Haslum for comments and suggestions on early versions of this paper, and Blai Bonet for providing us with updated versions of the HSP 2.0 planner. We also thank the reviewers for valuable comments and suggestions that helped increase the breadth and depth of the experimental evaluation.

References

[1] G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani. Bounded model checking for timed systems. In *Formal Techniques for Networked and Distributed Systems - FORTE 2002*, number 2529 in Lecture Notes in Computer Science, pages 243–259. Springer-Verlag, 2002.

[2] Gilles Audemard, Marco Bozzano, Alessandro Cimatti, and Roberto Sebastiani. Verifying industrial hybrid systems with MathSAT. *Electronic Notes in Theoretical Computer Science*, 119(2):17–32, 2005.

[3] Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97) and 9th Innovative Applications of Artificial Intelligence Conference (IAAI-97)*, pages 203–208, 1997.

[4] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[6] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[7] B. Bollobás. *Random graphs*. Academic Press, 1985.

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

[9] Blai Bonet, Gábor Loerincs, and Héctor Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97) and 9th Innovative Applications of Artificial Intelligence Conference (IAAI-97)*, pages 714–719. AAAI Press, 1997.

[10] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. The MathSAT 3 system. In *Automated Deduction - CADE-20*, volume 3632 of *Lecture Notes in Computer Science*, pages 315–321. Springer-Verlag, 2005.

[11] Tom Bylander. A probabilistic analysis of propositional STRIPS planning. *Artificial Intelligence*, 81(1-2):241–271, 1996.

[12] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In J. Mylopoulos, editor, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 331–337. Morgan Kaufmann Publishers, 1991.

[13] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via model checking: a decision procedure for $\mathcal{AR}$. In S. Steel and R. Alami, editors, *Recent Advances in AI Planning. Fourth European Conference on Planning (ECP'97)*, number 1348 in Lecture Notes in Computer Science, pages 130–142. Springer-Verlag, 1997.

[14] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[15] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81:31–57, 1996.

[16] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[17] Alfonso Gerevini and Lenhart Schubert. Inferring state constraints for domain-independent planning. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 905–912. AAAI Press, 1998.

[18] Alfonso Gerevini and Ivan Serina. Planning as propositional CSP: from Walksat to local search techniques for action graphs. *Constraints Journal*, 8:389–413, 2003.

[19] Nicolò Giorgetti, George J. Pappas, and Alberto Bemporad. Bounded model checking of hybrid dynamical systems. In *Proceedings of the 44th IEEE Conference on Decision and Control, and the European Control Conference 2005*, pages 672–677. IEEE, 2005.

[20] Carla P. Gomes and Bart Selman. Algorithm portfolio design: theory vs. practice. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI-97)*, pages 190–197. Morgan Kaufmann Publishers, 1997.

[21] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Journal of Artificial Intelligence Research*, 126(1–2):43–62, 2001.

[22] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97) and 9th Innovative Applications of Artificial Intelligence Conference (IAAI-97)*, pages 431–437. AAAI Press, 1998.

[23] C. P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1–2):67–100, 2000.

[24] Alban Grastien, Anbulagan, Jussi Rintanen, and Elena Kelareva. Diagnosis of discrete-event systems using satisfiability algorithms. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 305–310. AAAI Press, 2007.

[25] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

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

[27] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. A novel transition based encoding scheme for planning as satisfiability. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-10)*, pages 89–94, 2010.

[28] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.

[29] ICAPS. http://www.icaps-conference.org/, 2010.

[30] Matti Järvisalo and Tommi Junttila. Limitations of restricted branching in clause learning. *Constraints Journal*, 14:325–356, 2009.

[31] Henry Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363. John Wiley & Sons, 1992.

[32] Henry Kautz and Bart Selman. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201. AAAI Press, 1996.

[33] Henry Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In Thomas Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 318–325. Morgan Kaufmann Publishers, 1999.

[34] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In Martha Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 366–371. Morgan Kaufmann Publishers, August 1997.

[35] Nir Lipovetzky and Héctor Geffner. Inference and decomposition in planning using causal consistent chains. In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *ICAPS 2009. Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pages 217–224, 2009.

[36] Stephen M. Majercik and Michael L. Littman. Contingent planning under uncertainty via stochastic satisfiability. *Artificial Intelligence*, 147(1-2):119–162, 2003.

[37] J. P. Marques-Silva and K. A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[38] David A. McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the 9th National Conference on Artificial Intelligence*, volume 2, pages 634–639. AAAI Press / The MIT Press, 1991.

[39] Drew McDermott. The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Yale University, October 1998.

[40] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of SAT problems. In William Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 459–465. The MIT Press, 1992.

[41] David G. Mitchell. A SAT solver primer. *EATCS Bulletin*, 85:112–133, February 2005.

[42] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th ACM/IEEE Design Automation Conference (DAC'01)*, pages 530–535. ACM Press, 2001.

[43] Shougo Ogata, Tatsuhiro Tsuchiya, and Tohru Kikuno. SAT-based verification of safe Petri nets. In Farn Wang, editor, *Automated Technology for Verification and Analysis: Second International Conference, ATVA 2004, Taipei, Taiwan, ROC, October 31-November 3, 2004. Proceedings*, number 3299 in Lecture Notes in Computer Science, pages 79–92. Springer-Verlag, 2004.

[44] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving.* Addison-Wesley Publishing Company, 1984.

[45] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In Joao Marques-Silva and Karem A. Sakallah, editors, *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2007)*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer-Verlag, 2007.

[46] Aldo Porco, Alejandro Machado, and Blai Bonet. Automatic polytime reductions of NP problems into a fragment of STRIPS. In *ICAPS 2011. Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling*, pages 178–185, 2011.

[47] Katrina Ray and Matthew L. Ginsberg. The complexity of optimal planning and a more efficient method for finding solutions. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric Hansen, editors, *ICAPS 2008. Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, pages 280–287, 2008.

[48] Silvia Richter and Matthias Westphal. The LAMA planner: guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

[49] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Parallel encodings of classical planning as satisfiability. In José Júlio Alferes and João Leite, editors, *Logics in Artificial Intelligence: 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004. Proceedings*, number 3229 in Lecture Notes in Computer Science, pages 307–319. Springer-Verlag, 2004.

[50] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.

[51] Jussi Rintanen. A planning algorithm not based on directional search. In A. G. Cohn, L. K. Schubert, and S. C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR '98)*, pages 617–624. Morgan Kaufmann Publishers, 1998.

[52] Jussi Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

[53] Jussi Rintanen. Evaluation strategies for planning as satisfiability. In Ramon López de Mántaras and Lorenza Saitta, editors, *ECAI 2004. Proceedings of the 16th European Conference on Artificial Intelligence*, pages 682–687. IOS Press, 2004.

[54] Jussi Rintanen. Phase transitions in classical planning: an experimental study. In Didier Dubois, Christopher A. Welty, and Mary-Anne Williams, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR 2004)*, pages 710–719. AAAI Press, 2004.

[55] Jussi Rintanen. Asymptotically optimal encodings of conformant planning in QBF. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 1045–1050. AAAI Press, 2007.

[56] Jussi Rintanen. Regression for classical and nondeterministic planning. In Malik Ghallab, Constantine D. Spyropoulos, and Nikos Fakotakis, editors, *ECAI 2008. Proceedings of the 18th European Conference on Artificial Intelligence*, pages 568–571. IOS Press, 2008.

[57] Jussi Rintanen. Planning and SAT. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, number 185 in Frontiers in Artificial Intelligence and Applications, pages 483–504. IOS Press, 2009.

[58] Jussi Rintanen. Heuristics for planning with SAT and expressive action definitions. In *ICAPS 2011. Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling*, pages 210–217. AAAI Press, 2011.

[59] Jussi Rintanen. Engineering efficient planners with SAT. In *ECAI 2012. Proceedings of the 20th European Conference on Artificial Intelligence*, pages 684–689. IOS Press, 2012.

[60] Jussi Rintanen. Generation of hard solvable planning problems. Technical Report TR-CS-12-03, Research School of Computer Science, The Australian National University, March 2012.

[61] Jussi Rintanen. Heuristics for planning with SAT. In David Cohen, editor, *Principles and Practice of Constraint Programming - CP 2010, 16th International Conference, CP 2010, St. Andrews, Scotland, September 2010, Proceedings.*, number 6308 in Lecture Notes in Computer Science, pages 414–428. Springer-Verlag, 2010.

[62] Nathan Robinson, Charles Gretton, Duc-Nghia Pham, and Abdul Sattar. SAT-based parallel planning using a split representation of actions. In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *ICAPS 2009. Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pages 281–288. AAAI Press, 2009.

[63] Lawrence Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, September 2003.

[64] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-2004) and the 16th Conference on Innovative Applications of Artificial Intelligence (IAAI-2004)*, pages 337–343. AAAI Press, 1994.

[65] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 25:521–531, 1996.

[66] Andreas Sideris and Yannis Dimopoulos. Constraint propagation in propositional planning. In *ICAPS 2010. Proceedings of the Twentieth International Conference on Automated Planning and Scheduling*, pages 153–160. AAAI Press, 2010.

[67] Matthew Streeter and Stephen F. Smith. Using decision procedures efficiently for optimization. In Mark Boddy, Maria Fox, and Sylvie Thiébaux, editors, *ICAPS 2007. Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, pages 312–319, 2007.

[68] Vincent Vidal and Héctor Geffner. Branching and pruning: an optimal temporal POCL planner based on constraint programming. *Artificial Intelligence*, 170:298–335, 2006.

[69] Vincent Vidal. A lookahead strategy for heuristic search planning. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *ICAPS 2004. Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, pages 150–160. AAAI Press, 2004.

[70] Martin Wehrle and Jussi Rintanen. Planning as satisfiability with relaxed ∃-step plans. In Mehmet Orgun and John Thornton, editors, *AI 2007 : Advances in Artificial Intelligence: 20th Australian Joint Conference on Artificial Intelligence, Surfers Paradise, Gold Coast, Australia, December 2-6, 2007, Proceedings*, number 4830 in Lecture Notes in Computer Science, pages 244–253. Springer-Verlag, 2007.

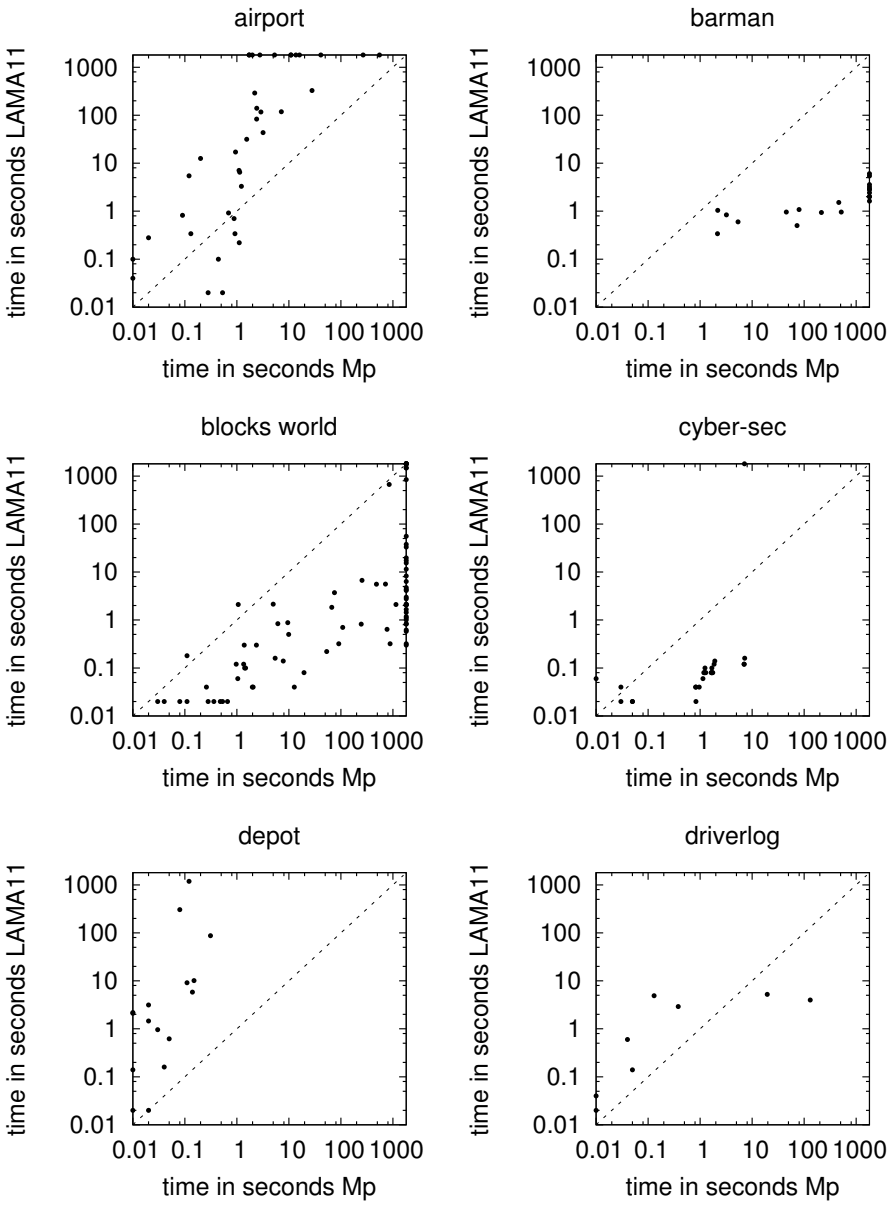
[71] Emmanuel Zarpas. Simple yet efficient improvements of SAT based bounded model checking. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings*, number 3312 in Lecture Notes in Computer Science, pages 174–185. Springer-Verlag, 2004.

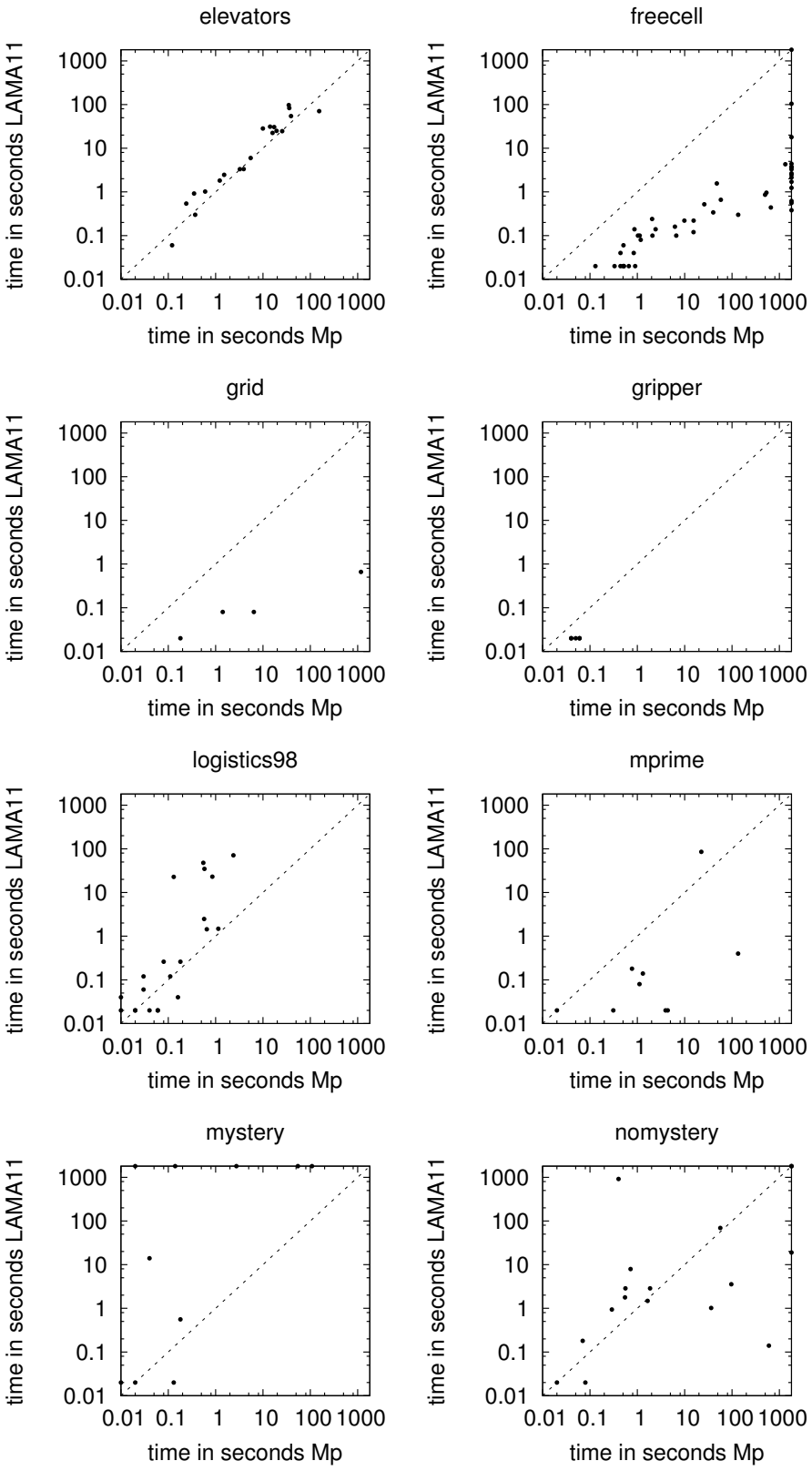
Appendix: Comparisons with Planning Competition Domains

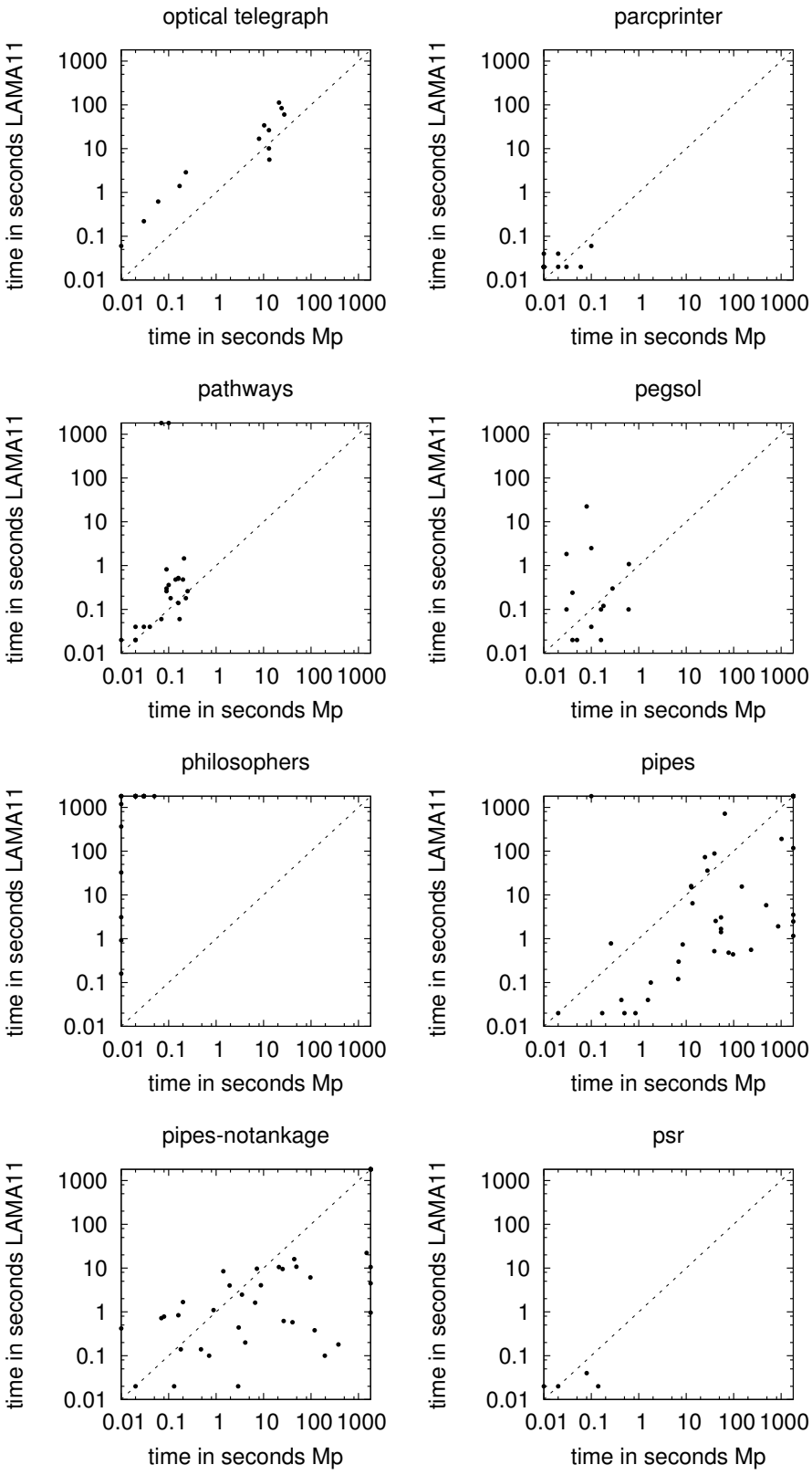
The diagrams in the next pages compare the runtimes and plan sizes of two planners instance by instance for each of the domains used in the planning competitions between 1998 and 2011. Some of the diagrams have fewer dots than indicated by Table 5. This is due to more than one instance having exactly the same runtimes for both planners, typically when the runtimes are close to 0 seconds.

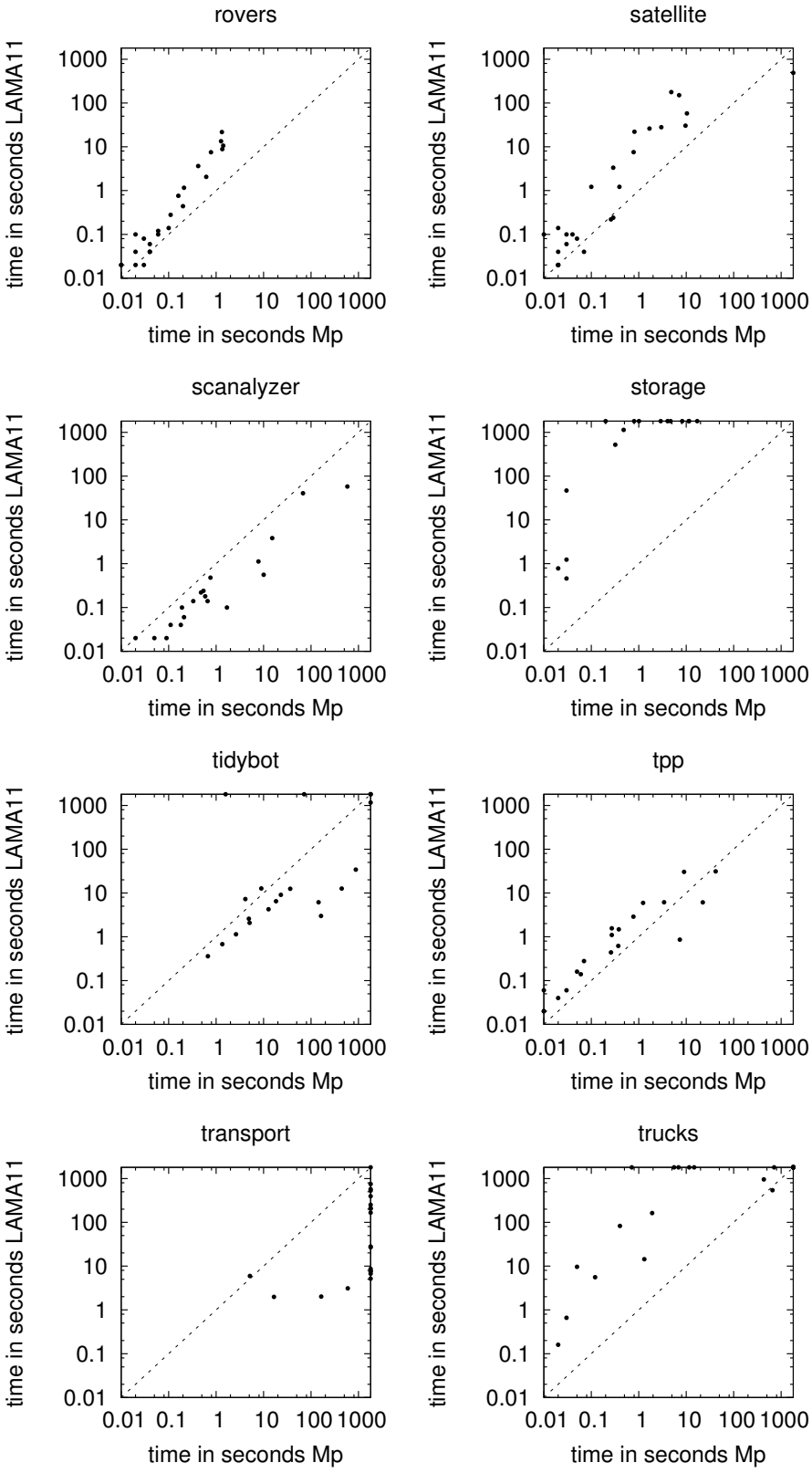
Comparison of Mp and LAMA Runtimes with STRIPS Benchmarks

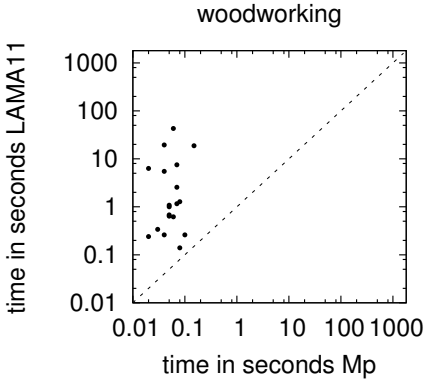
For this runtime comparison, we have only included the search times of both planner. The relatively long preprocessing times of LAMA11 and Mp are ignored, because both spend a lot of time finding invariants but by using radically different algorithms for this task. LAMA's preprocessor is generally slower but it scales up somewhat better than that of Mp to large instances.

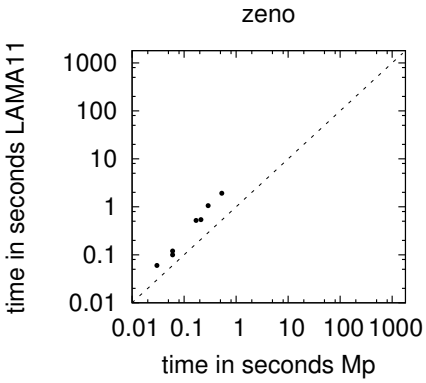



elevators
freecell
grid
gripper
logistics98
mprime
mystery
nomystery
time in seconds LAMA11
time in seconds Mp
0.01
0.1
1
10
100
1000

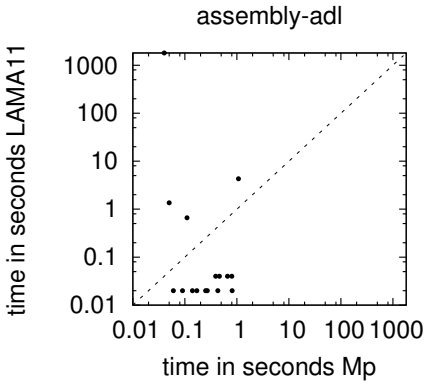

optical telegraph
parcprinter
pathways
pegsol
philosophers
pipes
pipes-notankage
psr
time in seconds LAMA11
time in seconds Mp
0.01
0.1
1
10
100
1000

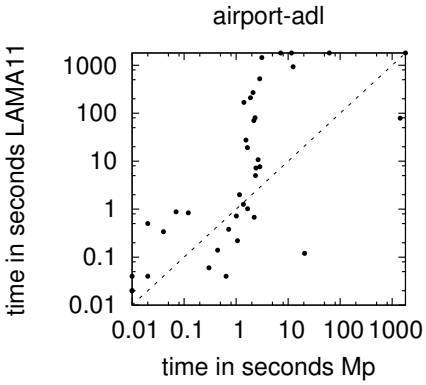

rovers
satellite
scanalyzer
storage
tidybot
tpp
transport
trucks
time in seconds LAMA11
time in seconds Mp
0.01
0.1
1
10
100
1000

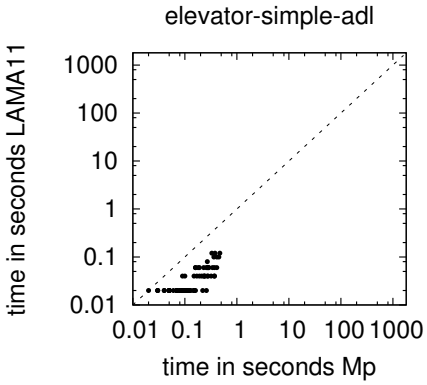


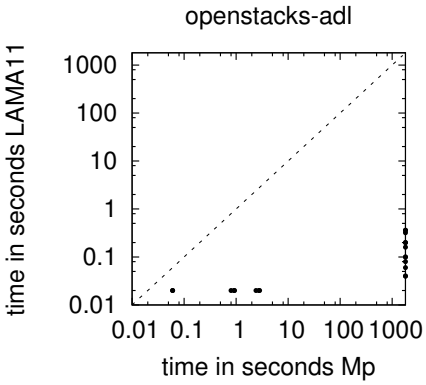


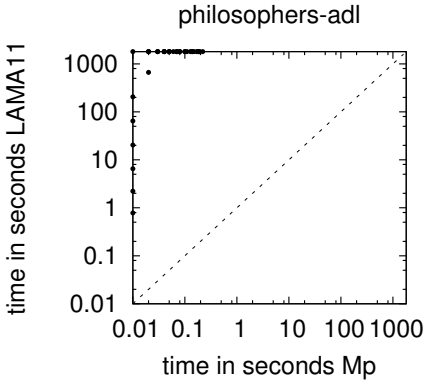
Comparison of Mp and LAMA Runtimes with ADL Benchmarks

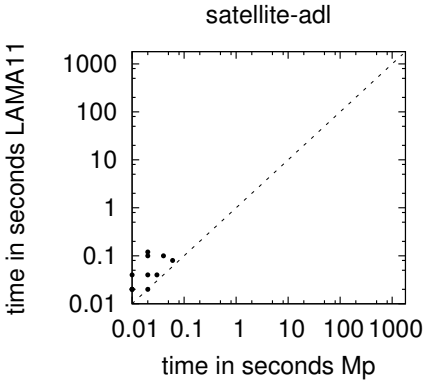


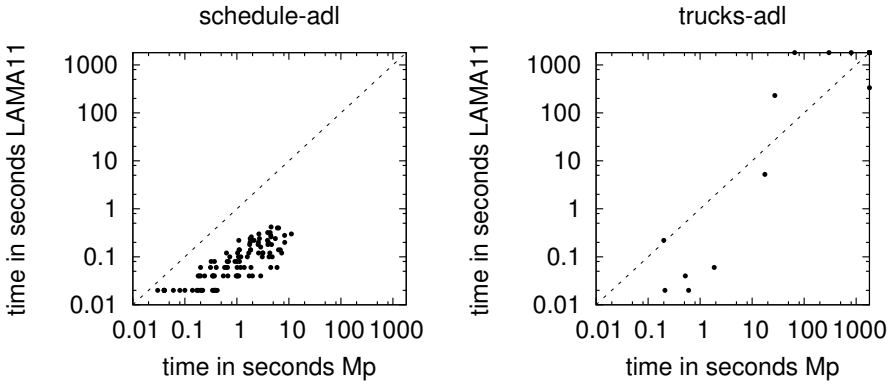




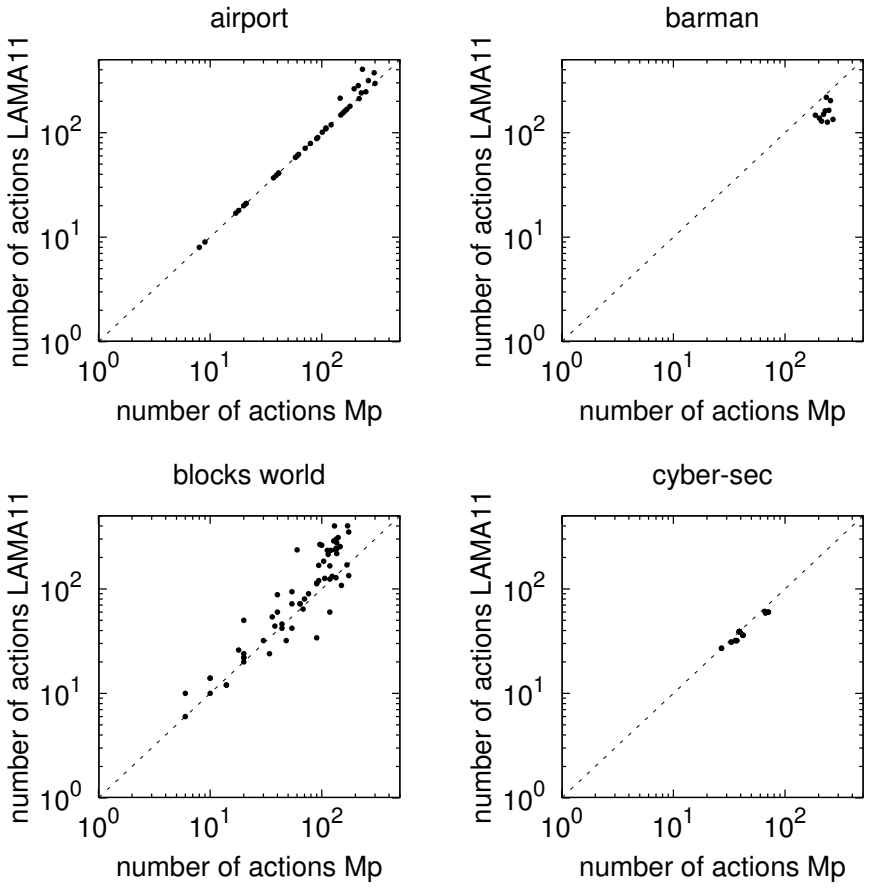


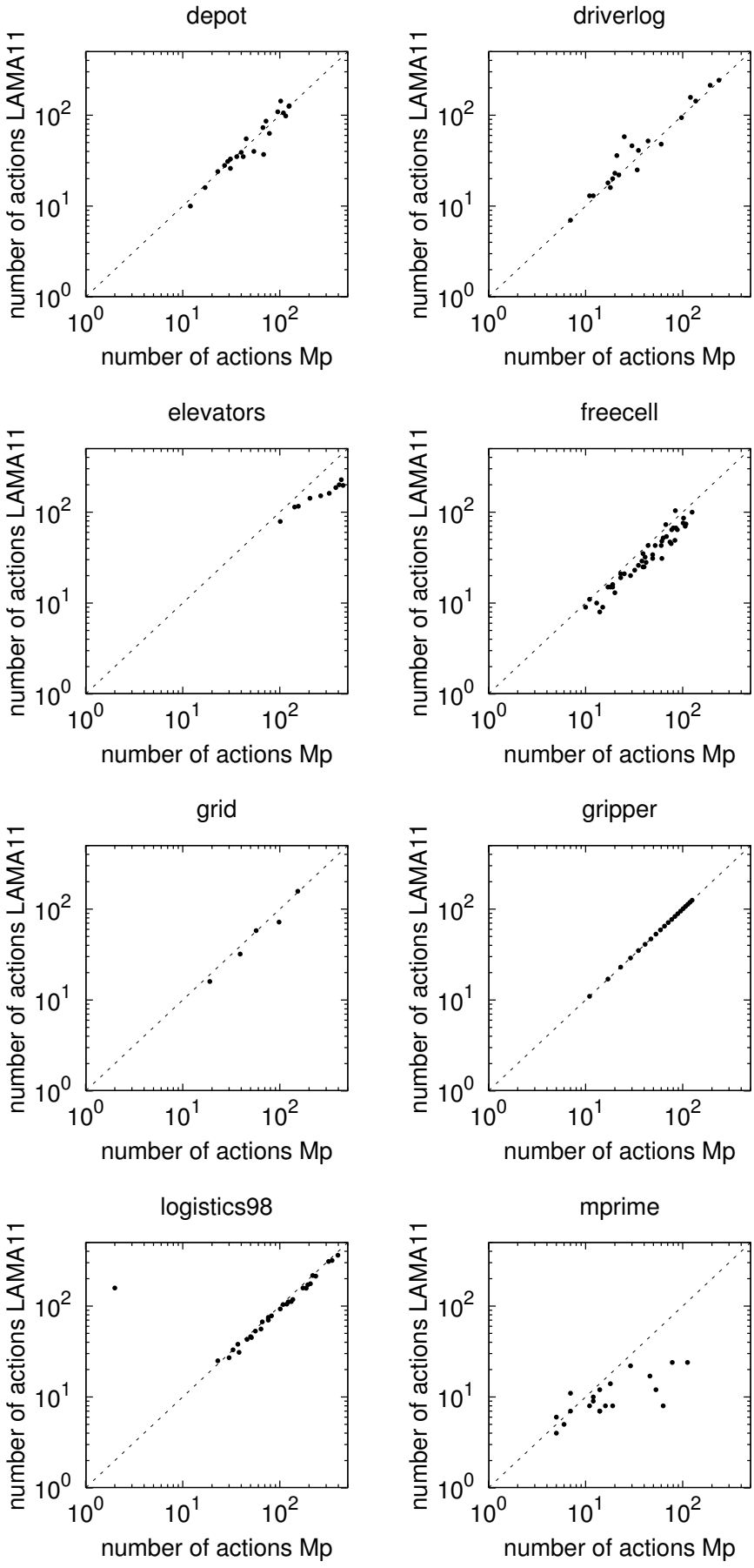


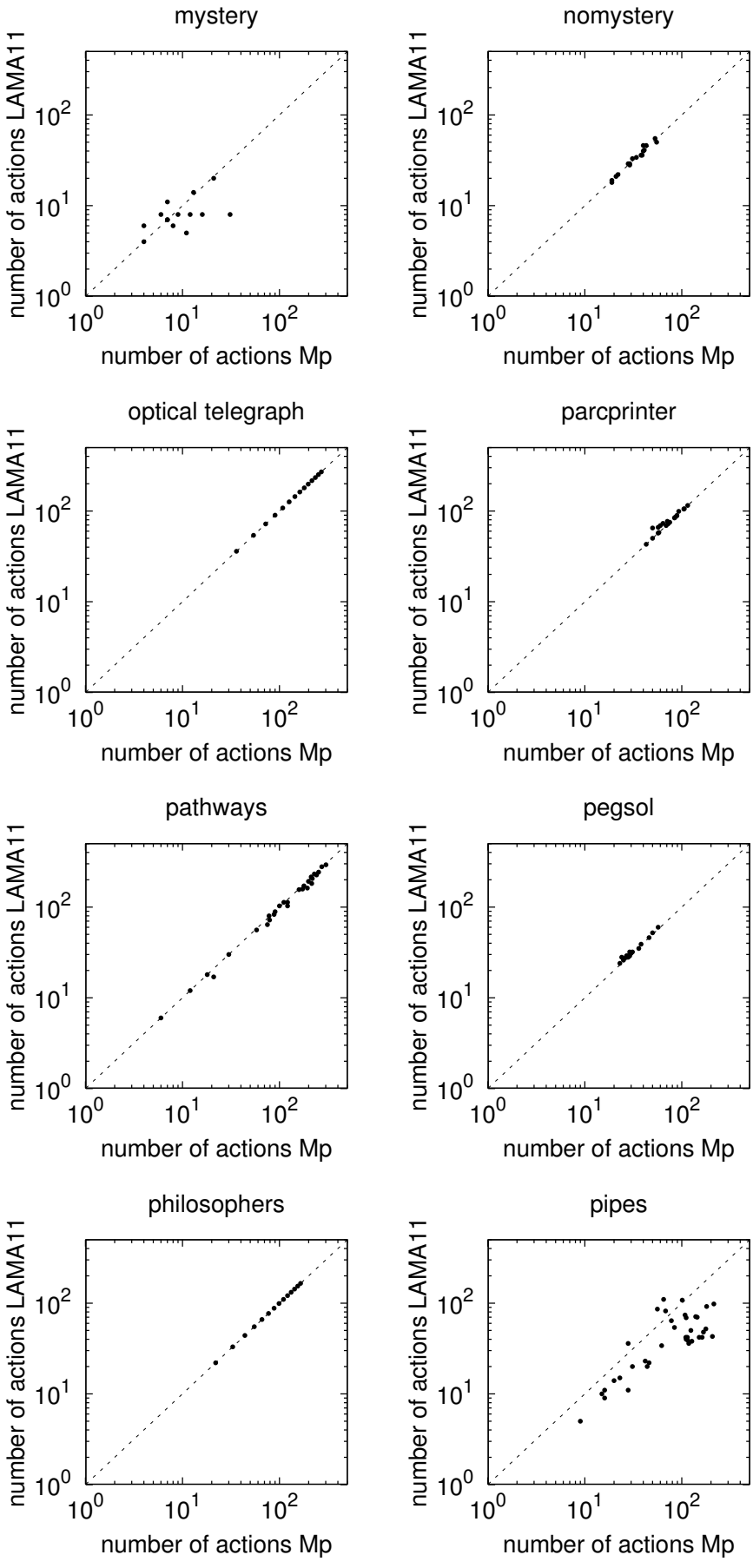


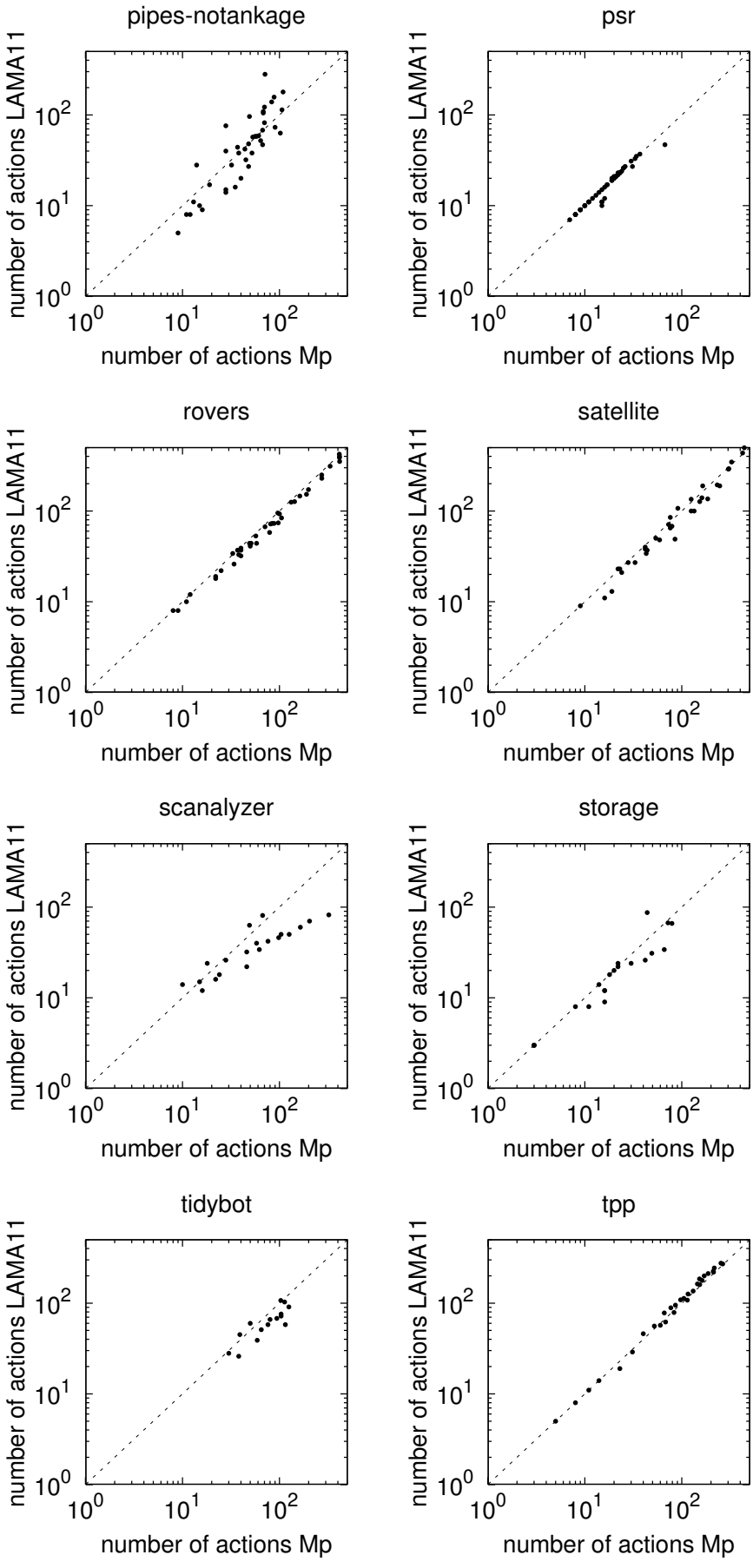


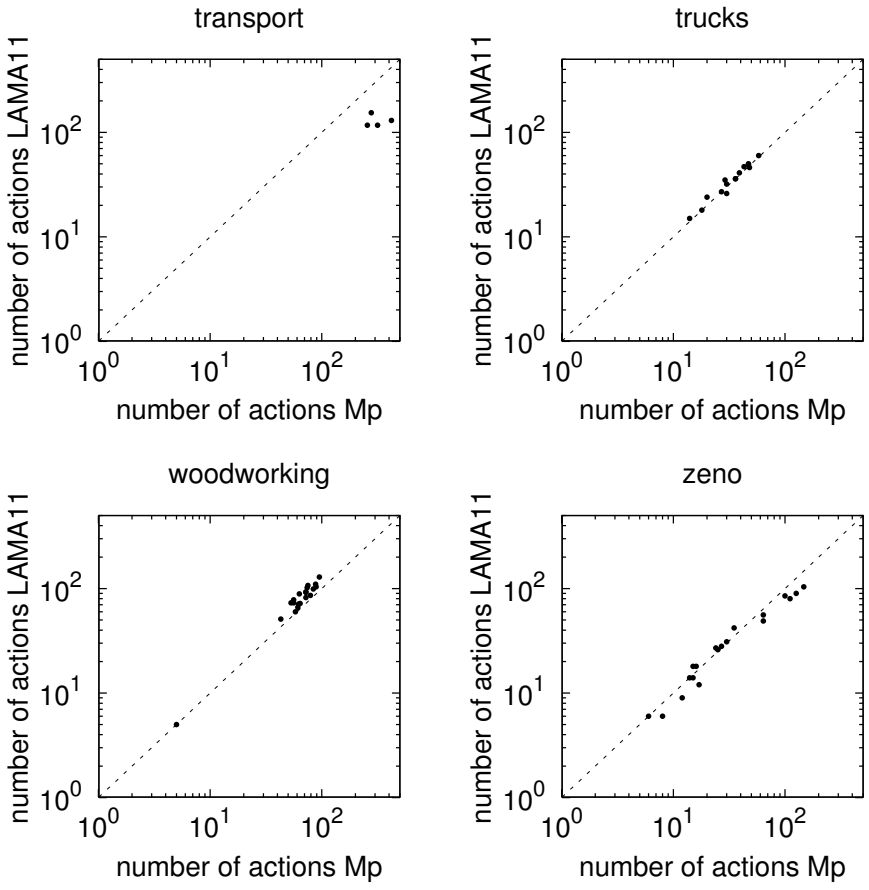
Comparison of Mp and LAMA Plan Sizes with STRIPS Benchmarks



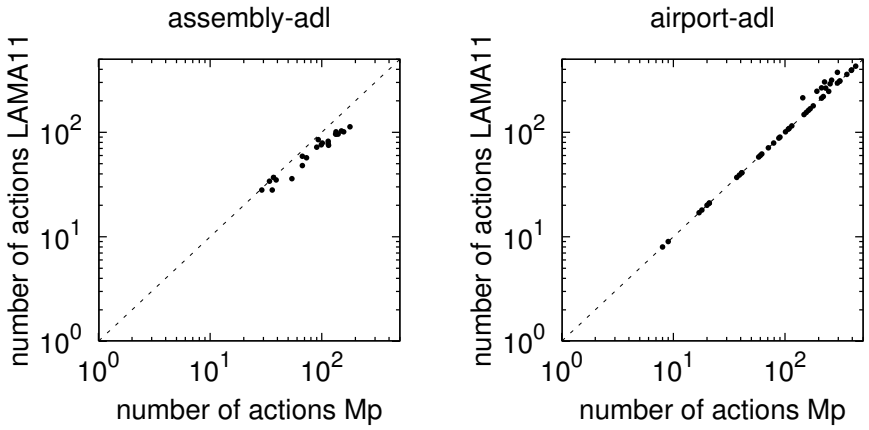

depot
driverlog
elevators
freecell
grid
gripper
logistics98
mprime
number of actions LAMA11
number of actions Mp
10^0
10^1
10^2

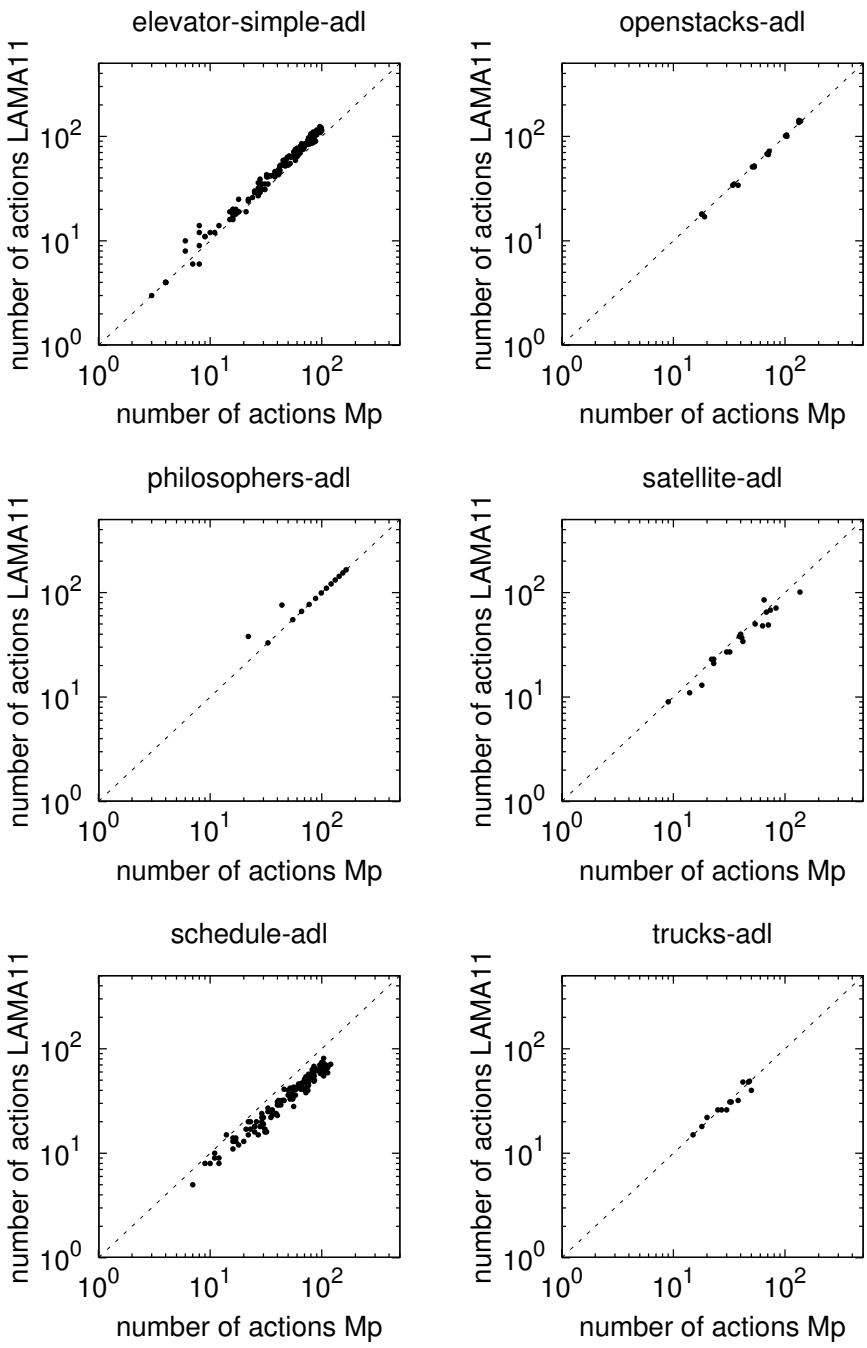

mystery
nomystery
optical telegraph
parcprinter
pathways
pegsol
philosophers
pipes
number of actions LAMA11
number of actions Mp
10^0
10^1
10^2


pipes-notankage
psr
rovers
satellite
scanalyzer
storage
tidybot
tpp
number of actions LAMA11
number of actions Mp
10^0
10^1
10^2

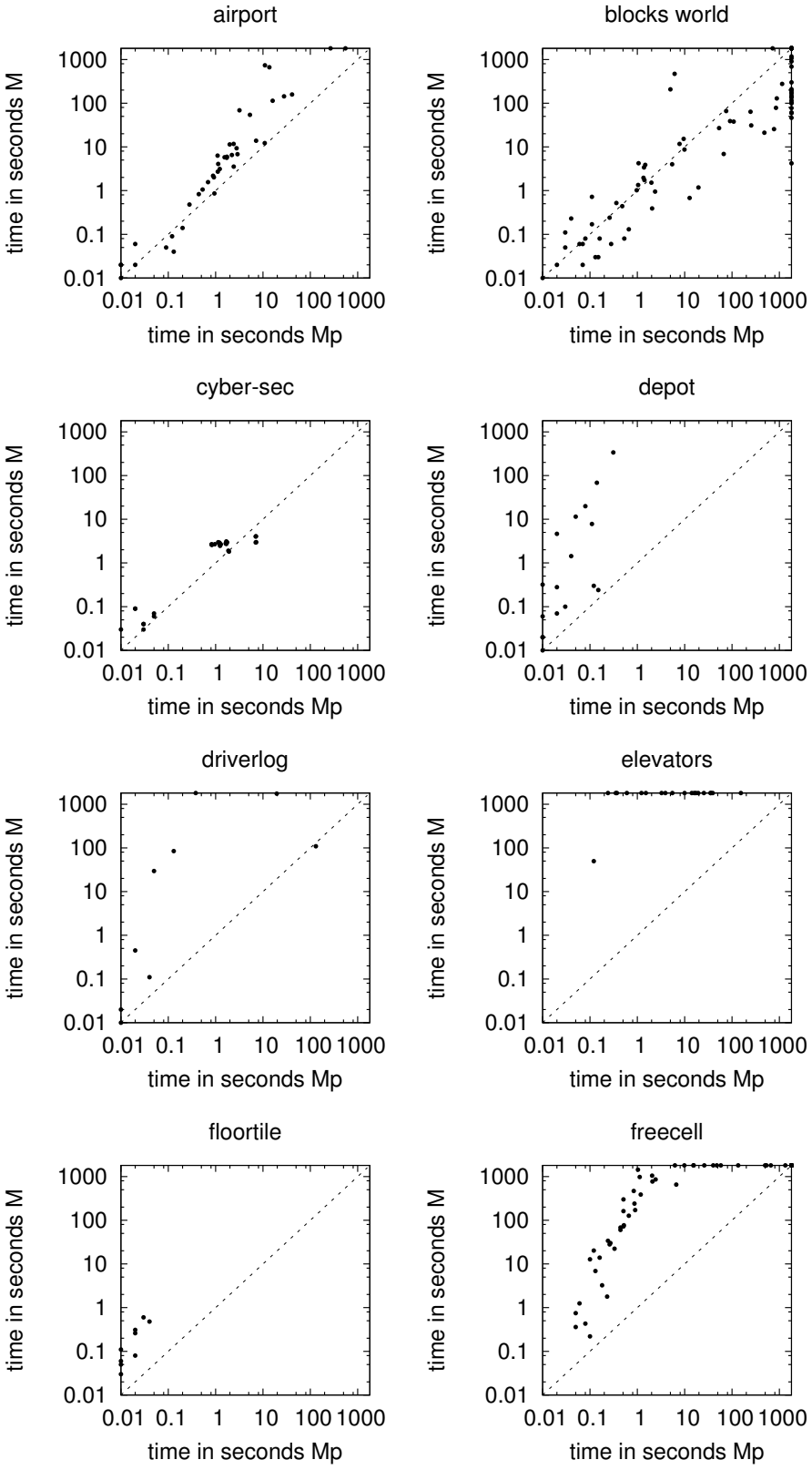


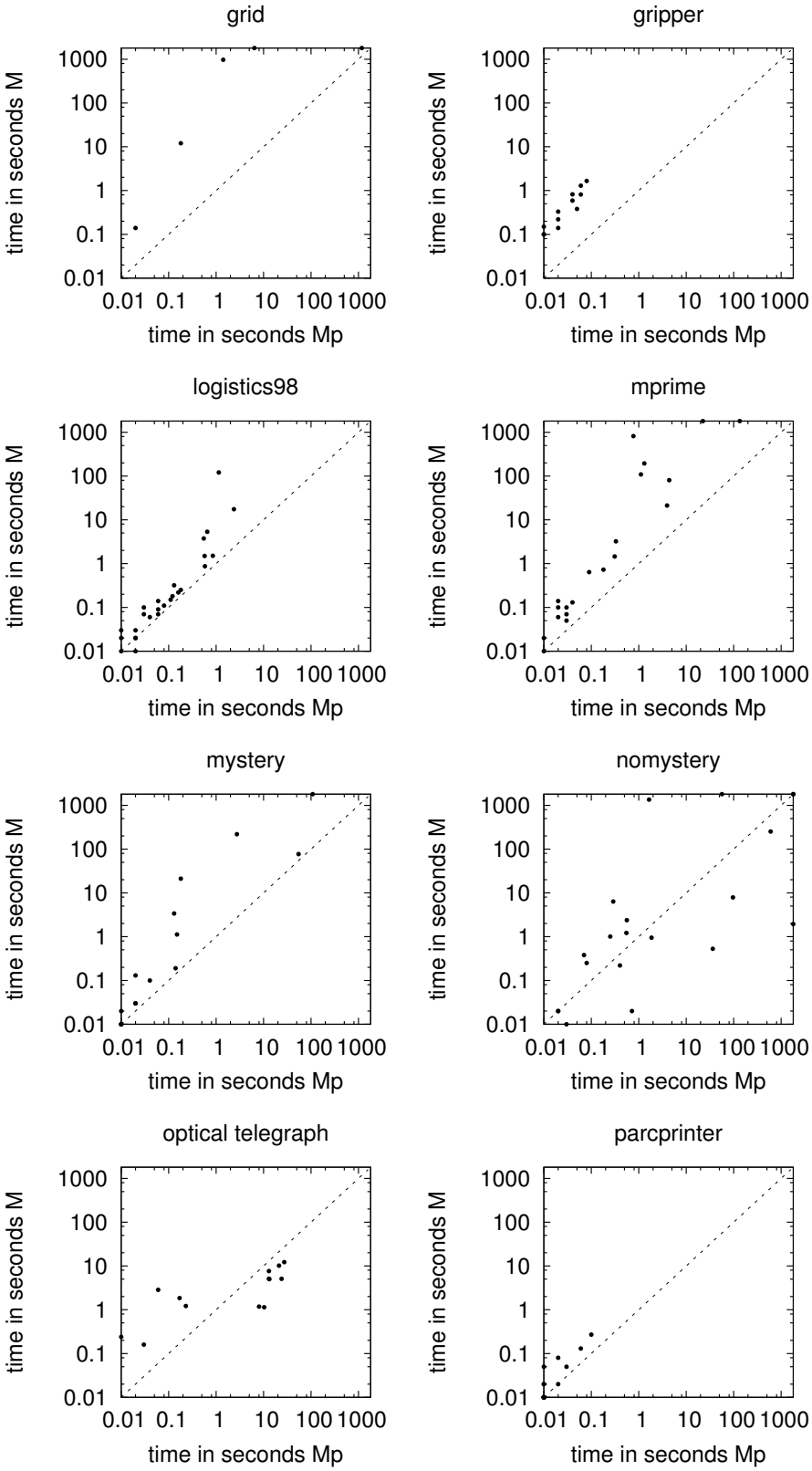
Comparison of Mp and LAMA Plan Sizes with ADL Benchmarks

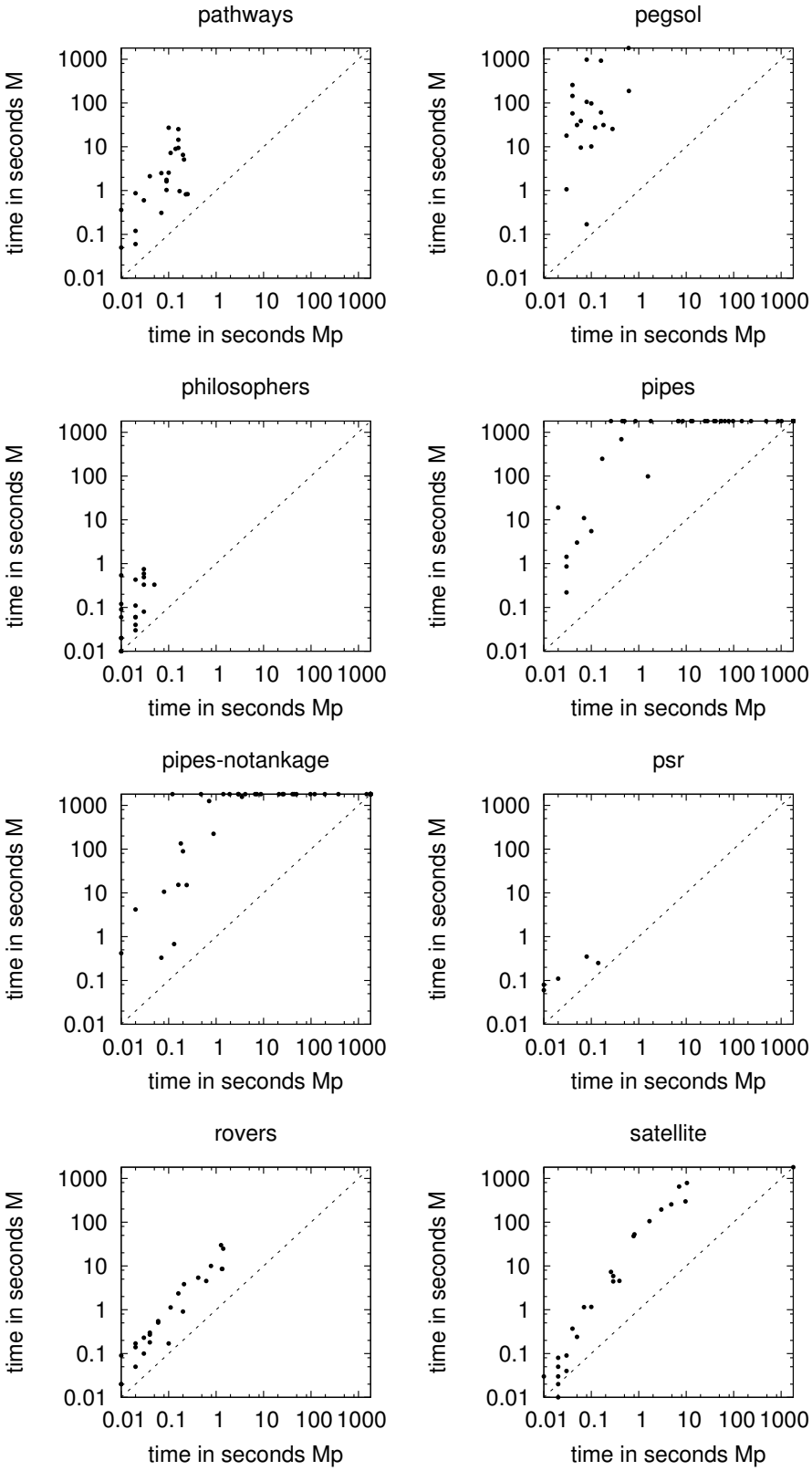


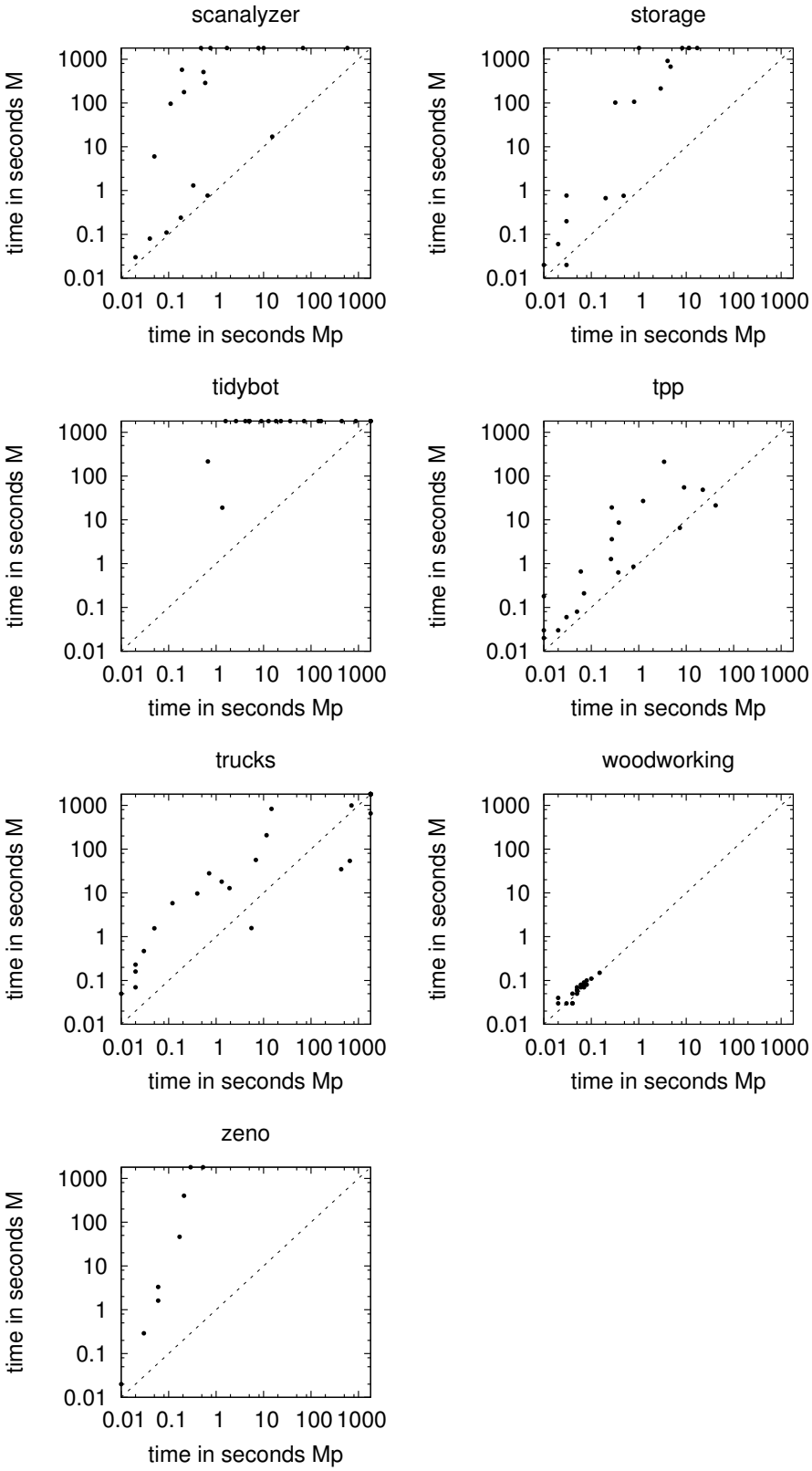

elevator-simple-adl
openstacks-adl
philosophers-adl
satellite-adl
schedule-adl
trucks-adl
number of actions LAMA11
number of actions Mp
10^0
10^1
10^2

Comparison of Mp and M Runtimes with STRIPS Benchmarks

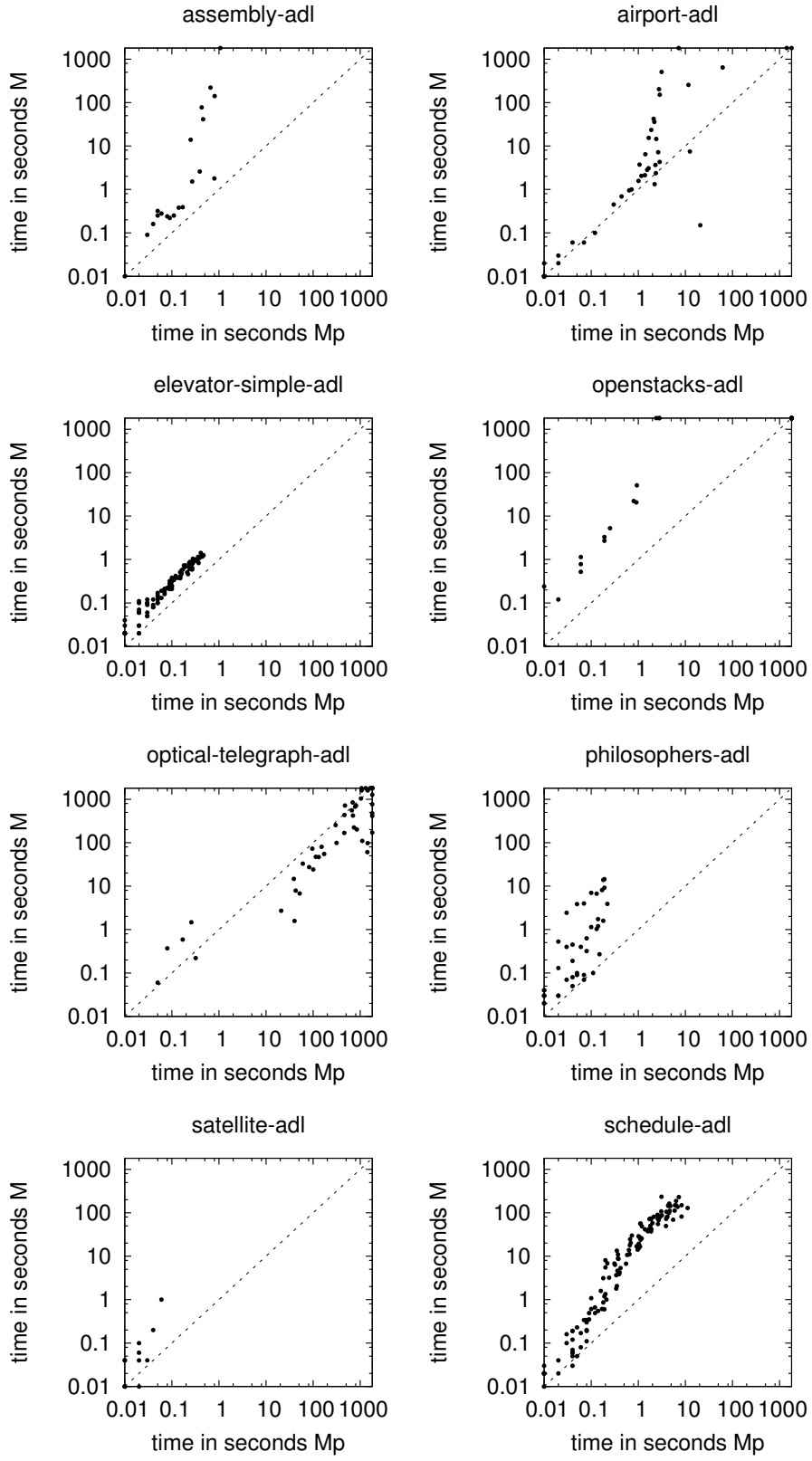


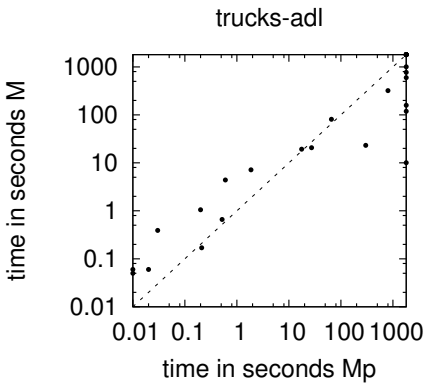

grid
gripper
logistics98
mprime
mystery
nomystery
optical telegraph
parcprinter
time in seconds M
time in seconds Mp
0.01
0.1
1
10
100
1000


pathways
pegsol
philosophers
pipes
pipes-notankage
psr
rovers
satellite
time in seconds M
time in seconds Mp
0.01
0.1
1
10
100
1000


scanalyzer
storage
tidybot
tpp
trucks
woodworking
zeno
time in seconds M
time in seconds Mp
0.01
0.1
1
10
100
1000

Comparison of Mp and M Runtimes with ADL Benchmarks




trucks-adl
time in seconds M
time in seconds Mp
1000
100
10
1
0.1
0.01
0.01
0.1
1
10
100
1000