Planning as Constraint Satisfaction: Solving the planning graph by compiling it into CSP

Minh Binh Do and Subbarao Kambhampati

Department of Computer Science and Engineering
Arizona State University, Tempe AZ 85287-5406
Email: {binhminh,rao}@asu.edu

Abstract

The idea of synthesizing bounded length plans by compiling planning problems into a combinatorial substrate, and solving the resulting encodings has become quite popular in recent years. Most work to-date has however concentrated on compilation to satisfiability (SAT) theories and integer linear programming (ILP). In this paper we will show that CSP is a better substrate for the compilation approach, compared to both SAT and ILP. We describe *GP-CSP*, a system that does planning by automatically converting Graphplan's planning graph into a CSP encoding and solving it using standard CSP solvers. Our comprehensive empirical evaluation of *GP-CSP* demonstrates that it is superior to both the Blackbox system, which compiles planning graphs into SAT encodings, and an ILP-based planner in a wide range of planning domains. Our results show that CSP encodings outperform SAT encodings in terms of both space and time requirements in various problems. The space reduction is particularly important as it makes *GP-CSP* less susceptible to the memory blow-up associated with SAT compilation methods. The paper also discusses various techniques in setting up the CSP encodings, planning specific improvements to CSP solvers, and strategies for variable and value selection heuristics for solving the CSP encodings of different types of planning problems.

[1] We thank Biplav Srivastava for explaining the inner workings of van Beek and Chen's constraint solver, and Terry Zimmerman for many useful comments on the earlier drafts of this paper. We also thank Peter van Beek for putting his CSP library in the public domain, and patiently answering our questions. This research is supported in part by NSF young investigator award (NYI) IRI-9457634, ARPA/Rome Laboratory planning initiative grant F30602-95-C-0247, AFOSR grant F20602-98-1-0182 and NSF grant IRI-9801676. The source code of the planner is available for downloading at http://rakaposhi.eas.asu.edu/gp-csp.html

1 Introduction

There are currently two dominant methods for solving planning problems. Planners such as Graphplan[2], SATPLAN[22], CPlan[35], and an ILP-based planner[36] follow the disjunctive planning approach while state-space planners such as FF[25] and HSP[4] use the conjunctive planning method. The disjunctive approach involves compiling a planning problem into a combinatorial substrate. Domain-independent planners working on the compilations to SAT and ILP (integer linear programming) have been studied in the literature. However, in this paper, we will argue that CSP is a better substrate than either SAT or ILP due to its rich structure and the flexibility to represent different types of constraints procedurally. Our experiments show that *GP-CSP*, a system that does planning by automatically converting Graphplan's planning graph into a CSP encoding and solving it using standard CSP solvers, is better than Blackbox, which compiles the planning problem into SAT, both in terms of running time and memory consumption on various planning problems. Given the fact that SAT based planners like Blackbox have been doing better than ILP based planners, we can claim that *GP-CSP* is the best among the planners using the compilation approach. Our results suggest that CSP-based approaches may also scale to planning in more expressive domains (e.g metric and temporal domains handled by the NASA EUROPA project[12]).

In this paper, we will consider CSP encodings that are automatically generated from Graphplan's planning graph structure. The decision to start with planning graph is due to the striking similarities between the backward search phase of Graphplan and constraint satisfaction problems [33] that have been pointed out by several researchers [19,37]. In most cases however, the detection of similarities has lead to adaptation of CSP techniques to Graphplan. For example, our own recent work [16,18] has considered the utility of adapting the explanation-based learning and dependency directed backtracking strategies from CSP to the backward search phase of Graphplan. More recently, researchers from CSP have shown interest in applying constraint programming to classical planning. Van Beek & Chen [35] describe a system called CPLAN that achieves impressive performance by posing planning as a CSP problem. However, an important characteristic (and limitation) of CPLAN is that it expects a hand-coded encoding; humans have to setup a domain and problem encoding independently for each problem and domain.

In this paper, we propose a different route to exploiting the similarities between the planning graph and CSP problems. We describe an implemented planner called *GP-CSP* (http://rakaposhi.eas.asu.edu/gp-csp.html) that extracts a solution from a planning graph by automatically converting it into CSP encodings. *GP-CSP* generates implicitly specified constraints wherever possible, to keep the encoding size small. The encoding is then passed onto the

standard CSP solvers in the CSP library created by van Beek[34]. Our empirical studies show that *GP-CSP* is superior to Graphplan and Blackbox, which compiles planning problems into SAT encodings, as well as ILP-encoding based planners [36]. While *GP-CSP*'s dominance over standard Graphplan in several domains is in terms of runtime, its advantages over Blackbox's SAT encodings include improvements in both runtime and *memory consumption*. The relative advantages of *GP-CSP* can be easily explained:

- Unlike the backward search in standard Graphplan, *GP-CSP* is not constrained by any directional search, and is able to to exploit all standard CSP search techniques straight out of the box. This involves non-directional search [29] as well as speedup techniques such as arc-consistency, dependency directed backtracking, explanation-based learning and a variety of variable ordering techniques. In practice, *GP-CSP* is found to be orders of magnitude faster than standard Graphplan on many benchmark problems.
- Compilation-based planning systems, such as SAT-based systems (Blackbox [21]) as well as an ILP-based system [36], are typically highly susceptible to memory blow-up [2] . CSP encodings used by *GP-CSP* are much less susceptible to this problem for two reasons. Since CSP allows multi-valued variables, while SAT considers only boolean variables, the SAT encoding of a problem tends to be larger than the CSP encodings. Second, *GP-CSP* is able to use implicitly (procedurally) specified constraints (c.f. [34]). This could keep the size of the encoding down considerably.
- CSP encodings also provide several structural advantages over SAT encodings. Typically, CSP problems have more structure than SAT problems, and we will argue that this structure can be exploited in developing search heuristics and learning algorithms that are suitable for planning encodings. Further, much of the knowledge-based scheduling work in AI is done by posing scheduling problems as CSP [40]. Approaches like *GP-CSP* may thus provide better substrates for integrating planning and scheduling. In fact, in related work [32], we discuss how CSP techniques can be used to tease resource scheduling away from planning. Specifically, *GP-CSP* has been used as one of the base planners in the RealPlan system, which integrates planners and schedulers to efficiently solve planning problems involving resources.

In addition to showing the relation between Graphplan and CSP search and a method for converting the planning graph into CSP, we also consider several approaches for improving the size and solvability of the CSP encodings. Specifically, our contributions include:

(1) A *compact encoding* that exploits the implicit CSP representation and

[2] Anecdotal evidence suggests that Blackbox's performance in the AIPS-98 planning competition was hampered mainly by its excessive memory requirements

significantly improves the size and solution time of the CSP encodings in many planning problems.

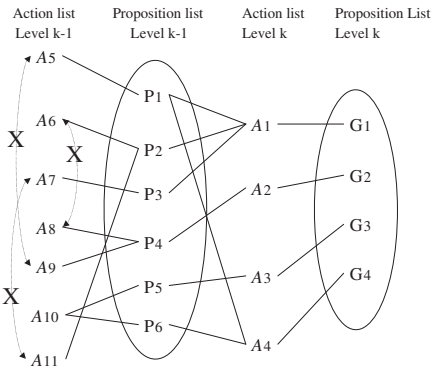
(2) Enhancements to the CSP solver by incorporating variations of explanation based learning (EBL) to improve the solution time of the CSP encodings.

(3) Investigations of different variable orderings to efficiently solve the CSP encodings of different types of planning problems. We also introduce a novel approach of automatically selecting variable ordering heuristics for solving a CSP encoding instance based on the analysis of the planning graph structure.

The rest of the paper discusses the design and evaluation of *GP-CSP*. In Section 2, we start with a brief review of Graphplan. Section 3 points out the connections between Graphplan and CSP, and discusses how the planning graph can be automatically encoded into a (dynamic) CSP problem. In Section 4, we describe the way *GP-CSP* automatically converts a planning graph into a CSP encoding in a format that is handled by the CSP library developed by van Beek[34]. Section 5 describes experiments that compare the performance of vanilla *GP-CSP* with standard Graphplan as well as Blackbox (with two different SAT solvers). We will consider improvements to the encoding size in Section 6 and in Section 7 we address improvements to the CSP solver, including the incorporation of EBL and investigation of different heuristics for variable and value orderings. Section 9 discusses the relation to other work and Section 10 summarizes the contributions of the paper and sketches several directions for future work.

2 Review of the Graphplan Algorithm

The Graphplan algorithm [2] can be seen as a "disjunctive" version of forward state space planners [19,15]. It consists of two interleaved phases – a forward phase, where a data structure called "planning graph" is incrementally extended, and a backward phase where the planning graph is searched to extract a valid plan. The planning graph consists of two alternating structures, called proposition lists and action lists. Figure 1 shows a partial planning graph structure. We start with the initial state as the zeroth level proposition list. Given a k level planning graph, the extension of the structure to level $k+1$ involves introducing all actions whose preconditions are present in the k^{th} level proposition list. In addition to the actions given in the domain model, we consider a set of dummy "persist" actions, one for each condition in the k^{th} level proposition list. A "persist-C" action has C as its precondition and C as its effect. Once the actions are introduced, the proposition list at level $k+1$ is constructed as just the union of the effects of all the introduced actions. The planning graph maintains the dependency links between the actions at



Variables: $G_1, \cdots, G_4, P_1 \cdots P_6$

Domains: $G_1{:}\{A_1\}, G_2{:}\{A_2\} G_3{:}\{A_3\} G_4{:}\{A_4\}$
$P_1{:}\{A_5\} P_2{:}\{A_6, A_{11}\} P_3{:}\{A_7\}$
$P_4{:}\{A_8, A_9\} P_5{:}\{A_{10}\} P_6{:}\{A_{10}\}$

Constraints (Mutex): $P_1 = A_5 \Rightarrow P_4 \neq A_9$
$P_2 = A_6 \Rightarrow P_4 \neq A_8$
$P_2 = A_{11} \Rightarrow P_3 \neq A_7$

Constraints (Activity): $G_1 = A_1 \Rightarrow Active\{P_1, P_2, P_3\}$
$G_2 = A_2 \Rightarrow Active\{P_4\}$
$G_3 = A_3 \Rightarrow Active\{P_5\}$
$G_4 = A_4 \Rightarrow Active\{P_1, P_6\}$

Init State: $Active\{G_1, G_2, G_3, G_4\}$

Fig. 1. A planning graph and the DCSP corresponding to it

level $k+1$ and their preconditions in level k proposition list and their effects in level $k+1$ proposition list. The planning graph construction also involves computation and propagation of "mutex" constraints. The propagation starts at level 1, by labeling as mutex the actions that are statically interfering with each other (i.e., their preconditions or effects are inconsistent). Mutexes are then propagated from this level forward by using two simple rules: two propositions at level k are marked mutex if all actions at level k that support one proposition are mutex with all actions that support the second proposition. Two actions at level $k+1$ are mutex if they are statically interfering or if one of the propositions (preconditions) supporting the first action is mutually exclusive with one of the propositions supporting the second action.

The search phase on a k level planning graph involves checking to see if there is a sub-graph of the planning graph that corresponds to a valid solution to the problem. This involves starting with the propositions corresponding to goals at level k (if all the goals are not present, or if they are present but a pair of them are marked mutually exclusive, the search is abandoned right away, and the planning graph is extended another level). For each of the goal propositions, we then select an action from the level k action list that supports it, such that no two actions selected for supporting two different goals are mutually exclusive (if they are, we backtrack and try to change the selection of actions). At this point, we recursively call the same search process on the $k-1$ level planning graph, with the preconditions of the actions selected at level k as the goals for the $k-1$ level search. The search succeeds when we reach level 0 (corresponding to the initial state).

Consider the (partial) planning graph shown on the left in Figure 1 that Graphplan may have generated and is about to search for a solution. $G_1 \cdots G_4$ are the top level goals that we want to satisfy, and $A_1 \cdots A_4$ are the actions that support these goals in the planning graph. The specific action-precondition dependencies are shown by the straight line connections. The actions $A_5 \cdots A_{11}$ at the left-most level support the conditions $P_1 \cdots P_6$ in the planning graph.

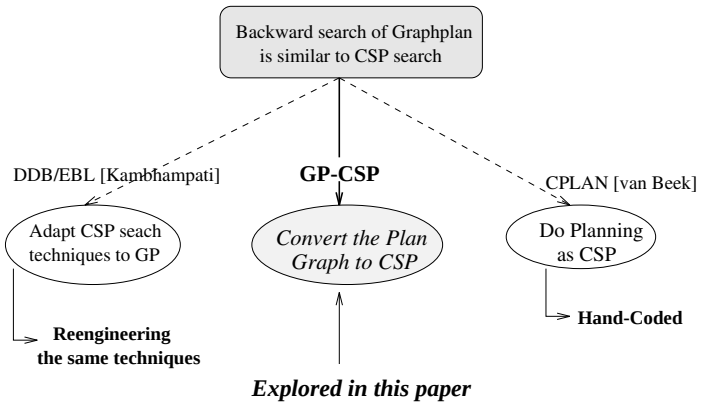


Fig. 2. *GP-CSP* framework in comparison with other efforts to take advantage of the similarity between Graphplan's backward search and the CSP search.

Notice that the conditions P_2 and P_4 at level $k-1$ are supported by two actions each. The x-marked connections between the actions A_5, A_9, A_6, A_8 and A_7, A_{11} denote that those action pairs are mutually exclusive. (Notice that given these mutually exclusive relations alone, Graphplan cannot derive any mutual exclusion relations at the proposition level k-1.)

3 Connections between Graphplan and CSP

Although the deep affinity between Graphplan's backward search and the process of solving constraint satisfaction problems has been noted earlier, these relations have hither-to been primarily used to adapt CSP search techniques into the backward search phase of Graphplan. Figure 2 shows how *GP-CSP* differentiates itself from other frameworks in making use of the similarity between the backward search of Graphplan and the CSP search.

The process of searching the planning graph to extract a valid plan from it can be seen as a dynamic constraint satisfaction problem (DCSP) [28]. The DCSP (also sometimes called a "conditional CSP" problem) is a generalization of the constraint satisfaction problem [33], that is specified by a set of variables, activity flags for the variables, the domains of the variables, and constraints on the legal variable-value combinations. In a DCSP, initially only a subset of the variables are active, and the objective is to *find assignments for all active variables that are consistent with the constraints among those variables.* In addition, the DCSP specification also contains a set of "activity constraints." An activity constraint is of the form: "if variable x takes on the value v_x, then the variables $y, z, w...$ become active."

The correspondence between the planning graph and the DCSP should now be clear. Specifically, the propositions at various levels correspond to the DCSP

variables [3], and the actions supporting them correspond to the variable domains. There are three types of constraints: *action mutex constraints, fact (proposition) mutex constraints* and *subgoal activation constraints.*

Since actions are modeled as values rather than variables, action mutex constraints have to be modeled indirectly as constraints between propositions.

If two actions a_1 and a_2 are marked mutex with each other in the planning graph, then for *every pair* of propositions p_{11} and p_{12} where a_1 is one of the possible supporting actions for p_{11} and a_2 is one of the possible supporting actions for p_{12}, we have the constraint:

$$\neg (p_{11} = a_1 \wedge p_{12} = a_2) \text{ or } p_{11} = a_1 \Rightarrow p_{12} \neq a_2$$

Fact mutex constraints are modeled as constraints that prohibit the simultaneous activation of the corresponding two facts. Specifically, if two propositions p_{11} and p_{12} are marked mutex in the planning graph, we have the constraint:

$$\neg (Active(p_{11}) \wedge Active(p_{12}))$$

Subgoal activation constraints are implicitly specified by action preconditions: supporting an active proposition p with an action a makes all the propositions in the previous level corresponding to the preconditions of a active.

Finally, only the propositions corresponding to the goals of the problem are "active" in the beginning. Figure 1 shows the dynamic constraint satisfaction problem corresponding to the example planning graph that we discussed in previous section.

There are two ways of solving a DCSP problem. The direct approach [28] involves starting with the initially active variables, and finding a satisfying assignment for them. This assignment may activate some new variables, and these newly activated variables are assigned in the second epoch. This process continues until we reach an epoch where no more new variables are activated (which implies success), or we are unable to give a satisfying assignment to the activated variables at a given epoch. In this latter case, we backtrack to a previous epoch and try to find an alternative satisfying assignment to those variables. The backward search process used by the Graphplan algorithm [2] can be seen as solving the DCSP corresponding to the planning graph in this direct fashion.

[3] Note that the same literal appearing in different levels corresponds to different DCSP variables. Thus, strictly speaking, a literal p in the proposition list at level i is converted into a DCSP variable p_i. To keep matters simple, the example in Figure 1 contains syntactically different literals in different levels of the graph.

Variables: $G_1, \cdots, G_4, P_1 \cdots P_6$	Variables: $G_1, \cdots, G_4, P_1 \cdots P_6$
Domains: G_1: $\{A_1\}, G_2$: $\{A_2\} G_3$: $\{A_3\} G_4$: $\{A_4\}$ P_1: $\{A_5\} P_2$: $\{A_6, A_{11}\} P_3$: $\{A_7\}$ P_4: $\{A_8, A_9\} P_5$: $\{A_{10}\} P_6$: $\{A_{10}\}$	Domains: G_1: $\{A_1, \perp\}, G_2$: $\{A_2, \perp\} G_3$: $\{A_3, \perp\} G_4$: $\{A_4, \perp\}$ P_1: $\{A_5, \perp\} P_2$: $\{A_6, A_{11}, \perp\} P_3$: $\{A_7, \perp\}$ P_4: $\{A_8, A_9, \perp\} P_5$: $\{A_{10}, \perp\} P_6$: $\{A_{10}, \perp\}$
Constraints (Mutex): $P_1 = A_5 \Rightarrow P_4 \neq A_9$ $P_2 = A_6 \Rightarrow P_4 \neq A_8$ $P_2 = A_{11} \Rightarrow P_3 \neq A_7$	Constraints (Mutex): $P_1 = A_5 \Rightarrow P_4 \neq A_9$ $P_2 = A_6 \Rightarrow P_4 \neq A_8$ $P_2 = A_{11} \Rightarrow P_3 \neq A_7$
Constraints (Activity): $G_1 = A_1 \Rightarrow Active\{P_1, P_2, P_3\}$ $G_2 = A_2 \Rightarrow Active\{P_4\}$ $G_3 = A_3 \Rightarrow Active\{P_5\}$ $G_4 = A_4 \Rightarrow Active\{P_1, P_6\}$	Constraints (Activity): $G_1 = A_1 \Rightarrow P_1 \neq \perp \wedge P_2 \neq \perp \wedge P_3 \neq \perp$ $G_2 = A_2 \Rightarrow P_4 \neq \perp$ $G_3 = A_3 \Rightarrow P_5 \neq \perp$ $G_4 = A_4 \Rightarrow P_1 \neq \perp \wedge P_6 \neq \perp$
Init State: $Active\{G_1, G_2, G_3, G_4\}$	Init State: $G_1 \neq \perp \wedge G_2 \neq \perp \wedge G_3 \neq \perp \wedge G_4 \neq \perp$

Fig. 3. Compiling a DCSP to a standard CSP

The second approach for solving a DCSP is to compile it into a standard CSP, and use the standard CSP algorithms. This compilation process is quite straightforward and is illustrated in Figure 3. The main idea is to introduce a new "null" value (denoted by "$\perp$") into the domains of each of the DCSP variables. We then model an inactive DCSP variable as a CSP variable which takes the value $\perp$. The constraint that a particular variable P be active is modeled as $P \neq \perp$. Thus, activity constraint of the form

$$G_1 = A_1 \Rightarrow Active\{P_1, P_2, P_3\}$$

is compiled to the standard CSP constraint

$$G_1 = A_1 \Rightarrow P_1 \neq \perp \wedge P_2 \neq \perp \wedge P_3 \neq \perp$$

It is worth noting here that the activation constraints above are only concerned with ensuring that propositions that are preconditions of a selected action take on non-$\perp$ values. They thus allow for the possibility that propositions can become active (take non-$\perp$ values) even though they are strictly not supporting preconditions of any selected action. Although this can lead to inoptimal plans, the mutex constraints ensure that no unsound plans will be produced [21]. To avoid unnecessary activation of variables, we need to add constraints to the effect that unless one of the actions needing that variable as a precondition has been selected as the value for some variable in the earlier (higher) level, the variable must have $\perp$ value. Such constraints are typically going to have very high arity (as they wind up mentioning a large number of variables in the previous level), and may thus be harder to handle during search. Therefore, we do not include those constraints in our implementation.

Finally, a mutex constraint between two propositions

$$\neg\left(Active(p_{11}) \wedge Active(p_{12})\right)$$

is compiled into

$$\neg\left(p_{11} \neq \perp \wedge p_{12} \neq \perp\right).$$

Since action mutex constraints are already in the standard CSP form, with this compilation, all the activity constraints are converted into standard constraints and thus the entire CSP is now a standard CSP. It can now be solved by any of the standard CSP search techniques [33] [4].

The direct method has the advantage that it closely mirrors the Graphplan's planning graph structure and its backward search. Because of this, it is possible to implement the approach on the planning graph structure without explicitly representing all the constraints. The compilation to CSP requires that planning graph be first converted into an extensional CSP. It does however allow the use of standard algorithms, as well as supporting non-directional search (in that one does not have to follow the epoch-by-epoch approach in assigning variables). This is the approach taken in *GP-CSP* [5].

3.1 Size of the CSP encoding

Suppose that we have an average of n actions and m facts in each level of the planning graph, and the average number of preconditions and effects of each action are p, and e respectively. Let s indicate the average number of actions supporting each fact (notice that s is connected to e by the relation $ne = ms$), and l indicate the length of the planning graph. For the *GP-CSP*, we need $O(lm)$ variables, and the following binary constraints:

- $O(ln^2e^2)$ binary constraints to represent mutex relations in action levels. To see this note that there are $O(ln^2)$ action mutex constraints in the planning graph. If two actions a_1 and a_2 are mutex and a_1 supports e propositions and a_2 supports e propositions, then we will wind up having to model this

[4] It is also possible to compile any CSP problem to a propositional satisfiability problem (i.e., a CSP problem with boolean variables). This is accomplished by compiling every CSP variable P that has the domain $\{v_1, v_2, \cdots, v_n\}$ into n boolean variables of the form $P\text{is}v_1 \cdots P\text{is}v_n$. Every constraint of the form $P = v_j \wedge \cdots \Rightarrow \cdots$ is compiled to *P-is-*$v_j \wedge \cdots \Rightarrow \cdots$. This is essentially what is done by the BLACKBOX system [21].

[5] Compilation to CSP is not a strict requirement for doing non-directional search. In [39], we describe a technique that allows the backward search of Graphplan to be non-directional (see the discussion in Section 10).

one constraint as $O(e^2)$ constraints on the legal values the propositions supported by a_1 and a_2 can take together.
- $O(lm^2)$ binary constraints to represent mutex relations in fact levels.
- $O(lmsp)$ binary constraints for activity relations.

In the default SAT encoding of Blackbox[21], we will need $O(l(m+n))$ variables (since that encoding models both actions and propositions as boolean variables), and the following constraints (clauses):

- $O(ln^2)$ binary clauses for action mutex constraints.
- $O(lm^2)$ binary constraints to represent mutex relations in fact levels.
- $O(lm)$ clauses of length s to describe the constraints that each fact will require at least one action to support it.
- $O(lnp)$ binary clauses to indicate that action implies its preconditions.

As the expressions indicate, *GP-CSP* has only $O(lm)$ variables compared to $O(l(n+m))$ in Blackbox's SAT encoding. However, the number of constraints is relatively higher in *GP-CSP*. This increase is mostly because there are $O(ln^2e^2)$ constraints modeling the action mutexes in *GP-CSP*, instead of $O(ln^2)$ constraints (clauses)[6]. The increase is necessary because in CSP, actions are not variables, and the mutual exclusions between actions has to be modeled indirectly as constraints on legal variable-value combinations.

The fact that direct translation of the planning graph into CSP leads to a higher number of constraints doesn't necessarily mean that *GP-CSP* will consume more memory than SAT encodings, however. This is because *GP-CSP* represents constraints in an implicit fashion, thus making for a more compact representation. In Section 6, we describe how much of this increase can be offset by exploiting the implicit nature of constraints in *GP-CSP*.

4 Implementation details of Compiling the Planning Graph to CSP

As mentioned in the previous section, *GP-CSP* uses a CSP encoding of the planning graph. The basic idea is to let Graphplan build the planning graph representation, and convert it into a CSP encoding, along the lines illustrated in Figure 3. Like Blackbox, the conversion process starts from the first level in which all the goals appear non-mutex with each other. If the CSP encoding is unsolvable, then we destroy the CSP encoding, extend the graph one more level and re-generate a new CSP encoding. We use the CSP library developed by

[6] Notice that all fact mutexes and action mutexes other than the static interference mutexes are redundant. Thus, they are not necessary to guarantee the correctness of the solution. They correspond to extra binary constraints that result from doing directional partial 2-consistency in the graph expansion phase.

van Beek[34], and thus our constraints are put in a format that is accepted by their library. Here are some implementation level details of the way encodings are generated:

(1) We start by removing all irrelevant nodes from the planning graph. This is done by essentially doing a reachability analysis starting from the goal propositions in the final level. This step reduces the size of the encoding so it only refers to the part of the planning graph that is actually relevant to solving the current problem.
(2) Each of the propositions in the minimized graph is given a unique CSP variable number, and the actions in the graph are given unique CSP value numbers.
(3) The domains of individual variables are set to the the set of actions that support them in the planning graph, plus one distinguished value corresponding to $\perp$ for all propositions in levels other than the goal level. The null value $\perp$ is placed as the first value in the domain of each variable.
(4) Setting up the constraints: van Beek's CSP library allows for the definition of implicit constraints. It does this by allowing the definition of schematized "constraint types" and declaring that a constraint of a particular type holds between a set of variables. Each constraint type is associated with a function that can check, given an assignment for the constraint variables, whether or not that constraint is satisfied by that assignment. In *GP-CSP*, we define three types of constraints called, respectively *activity constraints*, *fact mutex constraints* and *action mutex constraints*. The activity constraints just ensure that if the first variable has a non-null value, then the second variable should also have non-null value. The fact mutex constraints ensure that both of the variables cannot have non-$\perp$ values simultaneously. The action mutex constraints ensure that the values assigned for any two variables are not a pair of actions that are mutex with each other.
(5) Checking the constraints: The CSP formulation accepted by van Beek's CSP library is very general in the sense that it allows us to specify which variables participate in which constraint, and the type for each constraint, but nothing more. Unlike the explicit representation, in which the solver will automatically generate the set of satisfying or failure assignments given a set of constraints in the CSP formulation, we have to write customized checking functions for each type of constraint in the implicit representation. To make things easier for checking constraints, we create a global hashtable when setting up the CSP formulation. The hashtable maps the index of each individual constraint with the actual actions participating in that constraint. For the activity constraint, it is an action that when assigned for the fact at the higher level will cause the fact in the lower level to become active. For the mutex constraint, it is a pair of actions that are not allowed to be values of variables in that constraint. Whenever a constraint is checked by the solver, the corresponding

checking function will consult the hashtable to match the current values assigned for its variables with the values in the hash entry for that constraint, and return the value *true* or *false* accordingly.

5 Results with Initial Encodings

We have implemented *GP-CSP* as described above, and have compared its performance with other Graphplan based planning systems–including the standard Graphplan and Blackbox [21], which compiles the planning graph into a SAT encoding. Note that all three systems are based on the same original C implementation of Graphplan. Therefore, any differences in performance are solely due to their search time and conversion time. Furthermore, the time to convert the planning graph to CNF form in Blackbox, and to the CSP encoding in *GP-CSP* are similar, and are quite small compared with the graph expansion, and searching times. For example, in problem log-b, Blackbox spends 0.12sec for converting a graph, 1.1sec for expanding, and around 16.7sec for searching. For the same problem, our best *GP-CSP* implementation takes 0.11sec for conversion, 1.1sec for expanding the graph, and 2.79sec for solving the CSP encoding [7] . The CSP encodings are solved with GAC-CBJ, a solver that does generalized arc consistency and conflict-directed backjumping (DDB). This is the solver that the CPLAN system used [35].

Table 1 compares the performance of these systems on a set of benchmark problems taken from the literature. The *blocksworld, logistics* and *grids* problems are taken from the AIPS-98 planning competitions (the blocksworld and logistics domains also come with Blackbox planner's distribution). The parallel blocksworld domain is from the set of domains distributed with the HSP planner[4]. The results show that *GP-CSP* is competitive with Graphplan as well as Blackbox with two state-of-the-art solvers–SATZ and Relsat [8] . While there is no clear-cut winner for all domains, we can see that Graphplan is better for serial and parallel blocksworld domains, and worse for the logis-

[7] Note that we did not mention the time each of the two SAT solvers in Blackbox requires to convert the CNF form to their own internal structure. This extra time is not needed in our *GP-CSP* system, because we convert directly from the planning graph to the structure that the GAC-CBJ solver can use without using an intermediate form like the CNF formulation in SAT.

[8] To make comparisons meaningful, we have run the SATZ and Relsat solvers without the random-restart strategy, and set the cutoff-limit to 1000000000. This is mainly because random-restart is a technique that is not unique to SAT solvers; see for example [18] for the discussion of how a random-restart strategy can be employed in Graphplan. However, the running times of SAT solvers are still dependent on the initial random seeds, so we take an average of 10 runs for each problem.

	GP-CSP			Graphplan			Blackbox-Satz solver			Blackbox-Relsat solver		
	time(s)	mem	length	time(s)	mem	length	time(s)	mem	length	time(s)	mem	length
bw-12steps	7.59	11 M	12/12	0.42	1 M	12/12	8.17	64 M	12/12	3.06	70 M	12/12
bw-large-a	138	45 M	12/12	1.39	3 M	12/12	47.63	88 M	12/12	29.87	87 M	12/12
rocket-a	9.25	5 M	26/7	68	61 M	30/7	8.88	70 M	33/7	8.98	73 M	34/7
rocket-b	19.42	5 M	26/7	130	95 M	26/7	11.74	70 M	27/7	17.86	71 M	26/7
log-a	16.19	5 M	66/11	1771	177 M	54/11	7.05	72 M	73/11	4.40	76 M	74/11
log-b	2898	6 M	54/13	787	80 M	45/13	16.13	79 M	60/13	46.24	80 M	61/13
log-c	>3hours	-	-	>3 hours	-	-	1190	84 M	76/13	127.39	89 M	74/13
hsp-bw-02	1.94	5 M	10/4	0.86	1 M	10/4	7.15	68 M	11/4	2.47	66 M	10/4
hsp-bw-03	20.26	90 M	16/5	5.06	24 M	13/5	> 8 hours	-	-	194	121 M	17/5
hsp-bw-04*	814	262 M	18/6	19.26	83 M	15/6	> 8 hours	-	-	1682	154 M	19/6
grid-01	27.40	72 M	13/13	18.06	41 M	13/13	> 3 hours	112 M	-	31.78	118 M	14/13
grid-02	> 3 hours	101 M	-	22.55	62 M	14/14	> 3 hours	131 M	-	66.26	138 M	14/14
grid-03	337	147 M	15/15	27.35	68 M	15/15	> 3 hours	162 M	-	98.40	171 M	16/15

Table 1. Comparing direct CSP encoding of *GP-CSP* with Graphplan, and Blackbox. All problems except hsp-bw-04 were run on a Sun Ultra 5, 256 M RAM machine. Hsp-bw-04 was run on a Pentium 500 MHz machine running LINUX with 256 MB RAM. Time is in CPU seconds, mem is swap space in megabytes, length is the plan length of solutions in terms of actions/time-steps.

	FC	AC	Only A-Mutex (with FC)
bw-12steps	7.59	33.37	7.54
bw-large-a	138	1162	140
rocket-a	9.25	23.14	8.31
rocket-b	19.42	49.34	19.48
log-a	16.19	32.73	83
hsp-bw-02	1.94	6.81	1.83
hsp-bw-03	20.26	160	21.04

Table 2
Solving the CSP encoding with different local consistency enforcement techniques. FC, AC stand for Forward Checking and Arc-Consistency. Running time reported in cpu seconds.

tics, in which *GP-CSP* and the two SAT solvers do better. *GP-CSP* is quite competitive with the SAT solvers in most of the problems.

Of particular interest are the columns titled "mem" that give the amount of memory (swap space) used by the program in solving the problem. We would expect that *GP-CSP*, which uses implicit constraint representation, should take much less space than Blackbox which converts the planning graph into a SAT encoding. Several of the problems do establish this dominance. For example, most logistics problems take about 6 megabytes of memory for *GP-CSP*, while they take up to 80 megabytes of memory for Blackbox's SAT encoding. One exception to this memory dominance of *GP-CSP* is the parallel blocks world domain taken from the HSP suite [5]. Here, the inefficient way that the initial CSP encoding represents the mutex constraints seems to increase the memory requirements of *GP-CSP* as compared to Blackbox. In this domain, the number of actions that can give the same fact is quite high, which leads to a higher number of mutex constraints in the *GP-CSP* formulation, compared with SAT. Nevertheless, *GP-CSP* was still able to outperform both SATZ and Relsat in that domain in terms of time. Perhaps more surprising is the fact that *GP-CSP* consumes less memory in 4 problems than Graphplan, despite the fact that Graphplan extracts the solutions directly from the planning graph while *GP-CSP* uses an additional CSP encoding step. The primary reason for this is the memoization strategy used by standard Graphplan. The memos, while improving speed, also increase space consumption. The vanilla GAC-CBJ solver used in *GP-CSP* does not do any memoization (see Section 7.1 for the effect of adding nogood learning to *GP-CSP*)

The columns titled "length" in Table 1 give the length of the plans returned by each solver (both in terms of steps and in terms of actions). These statistics show that the solution returned by *GP-CSP* is strictly better or equal to Blackbox using either SATZ or Relsat for all tested problems in which both systems find a solution. However, for all but one problem, the standard directional backward search of Graphplan returns shorter solutions in terms of

the number of actions. This can be explained by noting that in the standard backward search, a proposition will be activated *if and only if* an action that needs that proposition as a precondition gets chosen in that search branch. In contrast, as we mentioned in Section 3, the activation constraints in the *GP-CSP* encoding only capture the *if* part, leaving open the possibility of propositions becoming active even when no action needing that proposition has been selected. This can thus lead to longer solutions. The loss of quality is kept in check by the fact that our default value ordering strategy first considers the "$\perp$" value for every variable (proposition), as long as it is not forbidden to consider that value.

We also did some preliminary experimentations to figure out the best settings for the solver, as well as the encoding. In particular, we considered the relative advantages of doing arc-consistency enforcement vs. forward checking, and the utility of keeping fact mutexes–which, as mentioned earlier, are derivable from action mutexes. Forward checking involves doing constraint propagation when all but one of the variables of a constraint are instantiated. Arc-consistency is more eager and attempts propagation even if two of the variables in the constraint are un-instantiated (since we only have binary constraints, propagation is always attempted). Table 2 shows the results of our study. The column titled "FC" shows the result of applying only forward checking for all 3 types of constraints, the column titled "AC" shows the result of using arc-consistency for all types of constraints. The comparison between these columns shows that forward checking is better in every problem. We thus went with forward checking as the default in all other experiments (including those reported in Table 1). The last column reports on the effect of removing the redundant fact mutex constraints from the encoding (assuming we are doing forward checking). Comparing this column with that titled "FC", we can see that while including fact mutex constraints in the encoding does not change the solving time for most of the tested problems, there is a problem (log-a) in which we can solve it 4 times faster if we include the fact mutex constraints. Because we have not found any problem in which fact mutex constraint considerably slows down the search or worsens memory consumption, we retain them in the default *GP-CSP* encoding.

6 Improving encoding size by exploiting implicit constraint representation

As mentioned earlier, the *GP-CSP* encoding described above models the mutex constraints in a way that is less compact than possible. A mutex constraint between two actions is translated to $O(e^2)$ constraints on the proposition-action (variable-value) combinations–leading to a total of $O(ln^2e^2)$ action mutex constraints.

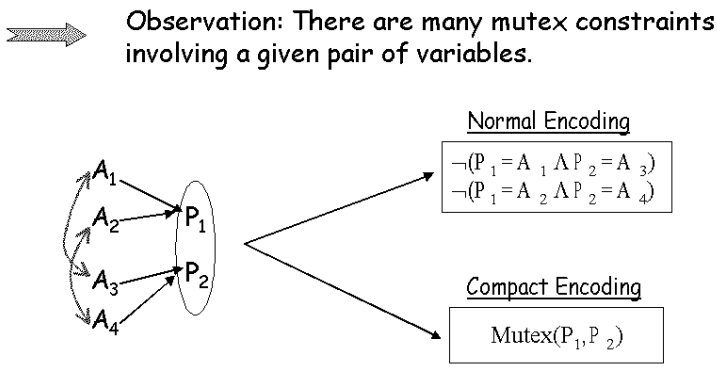


Fig. 4. Illustration of the differences between the Explicit and Implicit representation of mutex constraints.

We have devised a method that uses implicit constraint representation, and exploits the Graphplan data structures to reduce the number of constraints needed to model action and fact mutexes from $O(ln^2e^2)+O(lm^2)$ to $O(lm^2)$ (where m and n are, respectively, the average number of propositions and actions per level, l is the length of the planning graph, and e is the average number of effects of an action), while still keeping the arity of constraints binary. Figure 4 shows one example that demonstrates the differences between the explicit and implicit encodings of mutex constraints.

In contrast to the normal encoding, in which we start from a mutex relation between a pair of actions, and set up constraints over every pair of effects of those two actions, we will start from nodes in the fact levels for the compact encoding. For every pair of relevant facts in one level, we will check if at least one pair of actions supporting them are mutex. If there exists at least one such pair, we will set one mutex constraint involving those facts.

Notice that in the normal encoding, we commit to a specific action mutex whenever we set up a CSP mutex constraint, while we only have very general information about the relation between supporting actions in the compact encoding. In order to check the constraint, we will need a data structure that contains, for every pair of propositions, the list of forbidden action assignments for those propositions. In fact, Graphplan already keeps such a data structure, which is accessed with the function `are_mutex` in the standard implementation. Suppose that we have an action mutex constraint between facts P, and Q, and the current values assigned by the CSP solver to P, Q are a_1, a_2. We will use the call `are_mutex`(a_1,a_2) to check whether or not (a_1, a_2) are actually mutex for this particular action assignment. If they are, then we will tell the CSP solver to reject the current assignment.

Clearly, with this approach, there is only one type of mutex constraint and

	normal encoding					compact encoding				
	time	mem	length	mutex	total	time	mem	length	mutex	total
bw-12steps	7.59	11 M	12/12	96607	99337	1.96	3 M	12/12	6390	9120
bw-large-a	138	45 M	12/12	497277	503690	1234	11 M	12/12	26207	32620
rocket-a	9.25	5 M	26/7	21921	23147	4.01	3 M	26/7	4992	6218
rocket-b	19.42	5 M	26/7	26559	27881	6.19	4 M	26/7	5620	6942
log-a	16.19	5 M	66/11	16463	18392	3.34	4 M	64/11	4253	6182
log-b	2898	6 M	54/13	24301	26540	110	5.5 M	55/13	4149	6388
log-c	> 3 hrs	-	-	-	-	510	22 M	64/13	7772	3153
hsp-bw-02	1.94	5 M	10/4	78307	79947	0.89	4.5 M	11/4	2001	3641
hsp-bw-03	20.26	90 M	16/5	794670	800976	4.47	13 M	16/5	8585	14891
hsp-bw-04*	814	262 M	18/6	2892732	2907293	39.57	64 M	18/6	21493	36054
grid-01	27.40	72 M	13/13	160644	167623	23.58	62 M	13/13	8892	15871
grid-02	> 3 hours	101 M	-	278462	288135	36.70	82 M	14/14	13355	23028
grid-03	337	147 M	15/15	504635	518189	128	113 M	15/15	20467	34021

Table 3
Utility of encoding mutex constraints compactly. Time is in CPU seconds, *mem* is megabytes of swap memory, *mutex* is the number of CSP mutex constraints, and *total* is the total number of constraints in the CSP encoding.

the number of constraints needed to model action and fact mutexes is $O(lm^2)$, down from $O(ln^2e^2) + O(lm^2)$. This is because in the worst case, every pair of propositions at each level may be related by some action mutex.

Experiments with the new encoding show that it can help to reduce the number of CSP constraints needed for representing Graphplan's mutex relations from 4 to 140 times. Table 3 shows the comparison between the two types of encoding. The columns named "mutex" show that the number of CSP mutex-based constraints diminished by 4 to 140 times in the compact encoding, compared with the normal one. As a result, the memory consumed by *GP-CSP*, which is shown in the "mem" columns of Table 3, is reduced by 4 to 6 times for larger problems [9] . More importantly, *GP-CSP* now consumes less memory in all cases compared to Blackbox (see Table 1). The new encoding also seems to be easier to solve in all but one problem. In particular, problems log-b and hsp-bw-04 can be solved 28 and 20 times faster than under the normal encoding. For other problems the new encoding provides a speedup of up to 4x. The only problem that experiences considerable slowdown is bw-large-a, which is an easy problem to begin with. There is also not much difference in the length of the solutions returned. In the 3 problems where a difference occurs, the solution using the compact encoding is shorter by 2 actions in problem log-a and longer by 1 action in problems log-b and hsp-bw-02. Overall, the

[9] Note that the memory listed in Table 3 for *GP-CSP* includes the amount that is needed to hold the planning graph structure. The portion needed for the CSP encoding is in fact very small. For example, the memory needed for the CSP encoding of problems in the logistics domains ranges from 1 to 5 MB.

compact encoding is superior to the direct encoding and we make it our default encoding strategy for later experiments.

7 Improvements to the CSP Solver

The CSP solver that we have used for our initial experiments is the GAC-CBJ solver that comes pre-packaged with the CPLAN constraint library. GAC-CBJ uses forward-checking in conjunction with conflict directed backjumping. While this solver itself was quite competitive with Blackbox and Graphplan, we decided to investigate the utility of a variety of other enhancements commonly used to improve CSP solvers. The enhancements we investigated include: (1) explanation based learning (EBL) (2) level-based variable ordering, (3) distance based variable and value ordering [20], (4) automatic heuristics selection (5) min-conflict value ordering, and (6) the use of backward relevance-based mutex constraints [7]. In our experiments to-date, only the first four enhancements have demonstrated improvements in performance. We thus limit our discussion to these four enhancements.

7.1 EBL and nogood learning

The most important extension to the solver is the incorporation of EBL, which helps the solver to explain the failures it has encountered during search, and uses those explanations to avoid the same failures later [17]. The nogoods are stored as partial variable-value assignments, with the semantics that any assignment that subsumes a nogood cannot be refined into a solution. Extending GAC-CBJ to support EBL is reasonably straightforward as the conflict-directed backtracking already provides most of the required apparatus for identifying minimal failure explanations. Specifically, our nogood recording process is similar to the *jump-back learning* discussed in [13].

Once we know how to identify failure explanations, we have to decide how many explanations to store for future use. Indiscriminate storage of nogoods is known to increase both the memory consumption, and the runtime (in terms of the cost incurred in matching the nogoods to the current partial solution). Two of the best-known solutions for this problem in CSP are size-based learning [13], and relevance-based learning [1]. A k-degree size-based learning will not store any nogoods of size greater than k (i.e., any nogood which names more than k variables and their values). A k-degree relevance-based learning scheme will not store any no-good that differs from the current partial assignment in more than k variable-value assignments. Since relevance is defined with respect to the current partial assignment, relevance of a nogood

	size-based EBL						relevance-based EBL			
Problem	3-t	3-m	10-t	10-m	30-t	30-m	5-t	5-m	10-t	10-m
bw12steps	1.69	11M	**1.31**	11M	1.50	11	1.40	10M	1.46	10
bw-large-a	608	24M	259	24M	173	26M	**128**	24M	134	26M
rocket-a	3.69	8M	**2.08**	8M	2.49	8M	2.59	11M	2.39	11M
rocket-b	5.52	9M	**3.55**	9M	4.33	9M	4.31	10M	4.00	10M
log-a	2.67	15M	2.37	18M	**2.26**	18M	2.60	18M	2.30	18M
log-b	59.58	18M	39.55	19M	48.22	29M	36.77	18M	**35.13**	19M
log-c	153	24M	61	24M	63	34M	65	25M	**48.72**	25M
hsp-bw02	1.08	12M	**1.03**	12M	1.14	12M	1.05	12M	1.09	12M
hsp-bw03	5.08	26M	**5.04**	26M	5.19	26M	5.12	41M	5.17	41M
hsp-bw04*	40.41	86M	38.01	86M	24.07	89M	26.57	86M	**23.89**	86M
grid-01	23.26	68M	21.60	68M	**21.46**	68M	22.28	57M	21.98	62M
grid-02	36.78	92M	35.77	92M	**25.93**	92M	25.99	88M	26.69	92M
grid-03	124	127M	108	127M	58.68	127M	**53.51**	128M	53.68	128M

Table 4
Incorporating EBL into GAC-CBJ. Times are in CPU seconds. 3-t represents CPU times with nogoods limited to size 3 or less. 3-m gives the memory requirements in megabytes for these runs. Similar definitions hold for 5-t, 5-m, 10-t, 10-m, 30-t, and 30-m. All problems were run on a Sun Ultra 5 Unix machine with 256 MB of memory. To be consistent with other tables, problem hsp-bw04 was run on the Linux machine.

varies as we backtrack over partial assignments during search.

Table 4 shows the time and memory requirements in solving problems in blocksworld (serial and parallel), rocket, and logistics domains for both *size-based*, and *relevance-based* learning schemes. For size-based learning we experimented with size limits of 3, 10, and 30. The results suggest that the nogood size of around 10 gives the best compromise results between the time and memory requirements for most of the problems. However, for the two blocksworld domains, the bigger the size of nogoods we learn, the better the speedup we are able to get. In particular, for the parallel blocksworld domain, significant speedups only occur with $k \geq 30$.

For the relevance-based learning, we experimented with relevance limits of 5 and 10. In both cases, we also included a size limit of 50 (i.e., no nogood of size greater than 50 is ever stored, notwithstanding its relevance). The 4 columns grouped under the heading "relevance-based EBL" in Table 4 show the performance of relevance-based learning on *GP-CSP* in terms of time and memory consumptions. We see that relevance-based learning is generally faster than the best size-based learning in larger problems. The memory requirements for relevance and sized-based learning were similar. *We thus made relevance-10 learning to be the default in GP-CSP.* The performance of our default *GP-CSP* planner with various improvements will be presented in Section 8.

Problem	*GP-CSP*	Graphplan	SATZ	Relsat
bw12steps	1.34/0.30	0.28/0.1	5.60/6.40	2.10/7.00
bw-large-a	9.21/0.42	0.01/0.12	0.36/3.38	0.22/3.34
rocket-a	1.68/0.27	28.45/5.54	3.72/6.36	3.76/6.63
rocket-b	1.55/0.4	32.5/9.50	2.94/7.00	4.47/7.10
log-a	1.45/0.22	770/9.83	3.07/4.00	1.91/4.22
log-b	3.13/0.29	22.40/4.21	0.46/4.16	1.32/4.21
log-c	10.47/0.88	>220	24.42/3.36	2.61/3.56
hsp-bw02	0.82/0.37	0.79/0.08	6.56/5.67	2.27/5.50
hsp-bw03	0.86/0.32	0.98/0.59	>5570	37.52/2.95
hsp-bw04*	1.65/0.75	0.81/0.97	>1205	70.41/1.79
grid-01	1.07/1.00	0.82/0.66	>491	1.45/1.81
grid-02	1.38/0.89	0.85/0.67	> 405	2.48/1.5
grid-03	2.38/0.88	0.51/0.53	> 201	1.83/1.33

Table 5
The ratios of running time in CPU seconds and memory consumption in megabytes of *GP-CSP* using relevance-based EBL with nogood size limited to 10 (Rel-10 EBL) compared with naive *GP-CSP* (no EBL), Graphplan and Blackbox with two SAT solvers.

The four columns in Table 5 show the speedups in time, and the relative memory consumption of *GP-CSP* armed with relevance-10 EBL compared with the naive *GP-CSP* (with compact-encoding and no EBL), Graphplan, and Blackbox with SATZ and Relsat For example, the cell in the row named rocket-a, and the column titled Relsat has value 3.76/6.63. This means that *GP-CSP* with EBL is 3.76 times faster, and consumes 6.63 times *less* memory than Blackbox with Relsat on this problem. The results show that with EBL, the memory consumption of *GP-CSP* is increased, but it is still consistently 2 to 7 times smaller than Blackbox using both SATZ and Relsat solvers. *GP-CSP* with EBL is faster than Blackbox using Relsat (which is a powerful SAT solver, that basically uses the same search techniques as *GP-CSP*'s GAC-CBJ-EBL) in all but bw-large-a problem. It is slower than SATZ on only two problems, bw-large-a and log-b. The solution length, in terms of number of actions, returned by *GP-CSP* is also always smaller or equal to both SATZ and Relsat [10] .

7.2 Reusing EBL Nogoods across Encodings

Since for a given problem, the planning graph of size $k + 1$ is really a superset of the planning graph of size k, the CSP encodings corresponding to these two planning graphs have a considerable overlap. Indeed, Graphplan's

[10] The solutions returned by GAC-CBJ-EBL are always the same as the ones returned by GAC-CBJ

own backtrack search exploits the overlap between the encodings by reusing the failures ("memos") encountered in searching a k level planning graph to improve the search of a $k+1$ level planning graph. In contrast, *GP-CSP*, as discussed up to this point, does not exploit this overlap, and treats the encodings as essentially independent. (Blackbox too fails to exploit the overlap between consecutive SAT encodings).

Since inter-level memoization is typically quite useful for standard Graphplan, we also implemented a version of *GP-CSP* with EBL that exploits the overlap between consecutive encodings by storing the nogoods learned in a given encodings and reusing it in succeeding encodings. The main technical difficulty is that a nogood that is sound for the k^{th} level encoding may not remain sound for the $k+1^{th}$ level encoding. This might sound strange at first blush since the structure of the planning graph ensures that every variable in the k^{th} level encoding is also present, with identical domain and inter-variable constraints, in the $k+1^{th}$ level encoding. This fact should imply that a nogood made up of those variables must hold in the later encoding too. There is however one change when we go from one iteration to another; the specific variables that are "active" (i.e., must have non-$\perp$ values) change from level to level. Specifically, suppose the problem we are attempting to solve has a single top level goal G. In the k^{th} level encoding, the variable G_k, corresponding to the proposition G at level k will be required to have a non-$\perp$ value. However, when we go to $k+1^{th}$ level, the non-$\perp$ value constraint shifts to G_{k+1}, leaving G_k free to take on any value from its domain. Now, if there was a nogood N: $x_1 = v_1, .., x_i = v_i$ at the k^{th} level that was produced only because G_k was required to have a non-null value, N will no longer be sound in the $k+1^{th}$ level encoding.

Fortunately, there is a way of producing nogoods such that they will retain their soundness across successive encodings. It involves explicitly specifying the context under which the nogood holds. In the example above, if we remember the nogood N as $x_1 = v_1, .., x_i = v_i \wedge G_k \neq \perp$, then the contextualized nogood will be safely applicable across encodings. Producing such nogoods involves modifying the base-level EBL algorithm such that it tracks the "flaws" (variables with non-$\perp$ constraints) whose resolution forced the search down to specific failures, and conjoins them to the learned nogoods. In [17], Kambhampati provides straightforward algorithms for generating such contextualized nogoods, and we adapted those algorithms for *GP-CSP* [11] .

Although we managed to implement this inter-level nogood usage and verify its

[11] A second minor issue was to augment the program that compiles the planning graph into CSP encodings with some additional book-keeping information so that a proposition p at level l in the planning graph is conceptually mapped to the same CSP variable in all encodings.

	no-reuse (3)		no-reuse (10)		no-reuse (30)		reuse (3)		reuse (10)		reuse (30)	
	t(s)	mem	t(s)	m	t(s)	mem	t(s)	mem	t(s)	mem	t(s)	mem
bw-12steps	1.69	11MB	**1.31**	11MB	1.50	11MB	2.02	11MB	**1.37**	11MB	1.88	11MB
bw-large-a	608	24MB	259	24MB	**173**	26MB	1217	24MB	**400**	24MB	324	49MB
rocket-a	3.69	8MB	**2.08**	8MB	2.49	8MB	3.75	8MB	**2.57**	8MB	2.63	8MB
rocket-b	5.52	9MB	**3.55**	9MB	4.33	9MB	5.53	9MB	**3.95**	9MB	4.58	9MB
log-a	2.67	15MB	2.37	18MB	**2.26**	18MB	2.72	18MB	2.42	18MB	**2.30**	18MB
log-b	59.58	18MB	**39.55**	19MB	48.22	29MB	64	21MB	**48.94**	22MB	50.57	29MB
log-c	153	24MB	**61**	24MB	63	34MB	190	24MB	70	24MB	**64**	40MB
hsp-bw-02	1.08	12MB	**1.03**	12MB	1.14	12MB	1.21	12MB	**1.10**	12MB	1.21	12MB
hsp-bw-03	5.08	26MB	**5.04**	26MB	5.19	26MB	5.27	26MB	**5.16**	26MB	6.04	26MB
hsp-bw-04	82	88MB	73	88MB	**46.65**	91MB	82	88MB	74	88MB	**51.32**	92MB
hsp-bw-04*	40.41	86MB	38.01	86MB	**24.07**	89MB	40.68	86MB	37.96	86MB	**25.06**	90MB

Table 6. Reusing EBL nogoods across levels. The nogood learning strategy used in this experiment is k size-based EBL with k values of 3, 10, and 30. For each experiment, the two columns show the time in seconds, and memory consumptions in MB.

correctness, we found, to our disappointment, that reusing recorded nogoods does not provide a favorable cost-benefit ratio after all. We found that the use of such inter-level nogoods leads to consistently poorer performance than using intra-level nogoods alone in most of the problems. Table 6 compares the results of reusing EBL nogoods between consecutive encodings, and the default strategy of not reusing them.

There are several possible reasons as to why exploiting nogoods from previous levels did not lead to the improvements we expected. The most plausible explanation of this phenomenon is that it is caused by the differences between Graphplan's memoization strategy and the standard EBL nogoods (see [18]). In particular, as pointed out in [18], Graphplan's memos can be seen as nogoods of the form $P_1 \neq \bot \wedge \cdots \wedge P_j \neq \bot$ where P_i are all propositions from the same level of the planning graph. Such nogoods correspond to the conjunction of an exponential number of standard CSP nogoods of the form $P_1 = a_1 \wedge \cdots \wedge P_j = a_l$. Due to the allowance of inter-level nogoods, the total number of nogoods in *GP-CSP* increases more drastically than in Graphplan as we go to higher level encodings. As a result, the benefit from reusing nogoods in the previous encodings decreases, driving down the utility of storing and matching the previous level nogoods.

It is of course possible for us to increase the reusability of nogoods by concentrating only on the Graphplan-style abstract nogoods in the *GP-CSP* context. However, using such nogoods effectively requires that the search in *GP-CSP* be done level by level (akin to Graphplan)[12]. Unfortunately, as our experiments in the next section show, solving CSP encodings using a level by level (variable ordering) strategy is rarely the best choice for *GP-CSP*.

7.3 Utility of Specialized Heuristics for Variable and Value Ordering

7.3.1 Level-based Variable Ordering

Since standard Graphplan seems to do better than *GP-CSP* in domains like the serial blocksworld, we wondered if the level by level variable ordering, that is used in Graphplan, might also help *GP-CSP* to speed up the search in those domains. Currently, the GAC-CBJ solver used in *GP-CSP* uses *dynamic variable ordering* which prefers variables with smaller live domains (D), and breaks ties by the *most-constrained variable ordering* which prefers variables that take part in more constraints(C), followed by level-based variable ordering (L) which prefers variables from higher levels of the planning graph. Let us

[12] If we do not adopt a level by level approach here, the abstract nogood learning process will only terminate when the CSP search stopped. As a result, we can only learn one big nogood, which can only be used for the higher level encoding.

prob	LDC	LDC-E	DLC	DLC-E	DCL	DCL-E	GP
bw12steps	2.20	1.26	1.59	1.12	1.96	1.46	0.42
bw-large-a	12.90	6.88	13.24	6.58	1234	134	1.39
rocket-a	1240	52.12	4.71	2.29	4.01	2.39	68
rocket-b	629	43.23	118	15.82	6.19	4.00	130
log-a	>1800	>1800	>1800	22.93	3.34	2.30	1771
log-b	>1800	727	>1800	>1800	110	35.13	787
hsp-bw2	1.03	1.08	1.12	1.05	0.89	1.09	0.86
hsp-bw3	5.21	5.23	5.18	5.12	4.47	5.17	5.06
hsp-bw4*	5.76	4.87	19.29	14.64	39.57	23.89	19.26
grid-01	18.70	19.71	22.03	20.24	23.58	21.98	18.06
grid-02	24.02	24.58	25.46	24.63	36.70	26.69	22.55
grid-03	30.51	31.23	38.80	35.10	128	53.68	27.35

Table 7
GP-CSP with different variable orderings. The notations for different heuristics are: D: dynamic variable ordering, C: most (statically) constrained variable ordering, L: late-level first variable ordering, E: with EBL. The EBL used in this experiment is size-based EBL with maximum nogood size is set to 10. DCL is the default variable ordering of the CSP solver that we have been using so far. All experiments were run on the Ultra5 Unix machine, except hsp-bw4, which was run on Linux 500MHz machine. Running times are in CPU seconds.

call this default strategy the DCL strategy. DCL strategy gives tertiary importance to the planning graph level information. To make variable ordering more Graphplan-like, we tried two other variable orderings LDC, which gives primary importance to planning graph level, and DLC which gives it secondary importance. The performance of these three variable ordering strategies are compared in Table 7. As we can easily see, the new variable orderings significantly speedup *GP-CSP* in the two blocksworld domains, but slows the search down in the logistics domain. Even though EBL helps to speedup the LDC variable ordering by up to more than 20 times in the logistics domain, it is still significantly slower than the other two options.

7.3.2 HSP-based Variable and Value Ordering

The results of previous section suggest that simple variable ordering schemes such as DVO are not always effective for CSP encodings of planning problems. Variable and value ordering heuristics more suited to planning problems in different classes of domains are thus worth investigating. In this section, we will describe a variation of the variable and value ordering heuristics [13] used in

[13] Because the backward search of Graphplan proceeds backward level by level, and chooses one set of actions at a time, the heuristics discussed in [20] are based on the values applied to a set of propositions. However, the search in our planner is done for one variable at a time. Therefore, we have to modify the calculation and usage

the Graphplan algorithm that is discussed in [20]. These heuristics are based on the *difficulty* of achieving propositions in the planning graph, or alternatively their distance from the initial state, as measured by the number of action applications needed to achieve them from the initial state. The distance of each fact node in the graph is approximated by the first level that it appears in the planning graph structure. The difficulty in achieving a set of propositions is computed in terms of the distances of all the individual propositions (either by a SUM or MAX operation; see below). We call these distance estimates "hsp-values" after the terminology in [4,20]. Specifically, we compute the hsp-values for all fact and action nodes in the graph as follows:

- The hsp-value of each fact is the value of the first level in which it appears in the planning graph.
- The hsp-value of each action is the maximum value (max-hsp), or the sum (sum-hsp) of the hsp-values of its preconditions.

The hsp-values of the fact nodes will be used to setup the variable ordering, and the hsp-values of the action nodes will be used for value ordering in our CSP search. We still follow CSP's basic strategy of choosing the most difficult variable (in assigning value) first, and the least constrained value first. However, the difficulty here is not measured by the number of remaining values in a variable's domain (DVO), or the number of constraints that a variable participates in, but by the approximate distance (number of actions) to the initial state. More specifically, the search involves:

- Choosing the CSP variable corresponding to the fact node with highest hsp-value. Ties are broken by the normal most constrained variable ordering heuristic.
- For a given variable, choosing the CSP value in its domain corresponding to the action with smallest hsp-value.

Table 8 shows the results of using the hsp-values for variable and value orderings on a set of benchmark problems. The column titled MAX shows the results of the max-hsp value ordering, and the column SUM shows the results of using the sum-hsp value ordering. "Default" is the normal DCL variable ordering used in the GAC-CBJ solver (refer to Section 7.3.1). As suggested in [20], we tested with two cases for each problem: normal case, in which we start searching from the level that all the goals first appear non-mutex with each other, and the *skip* case, in which we skip the levels that do not contain the solution and start from the first solution-bearing level. The results show that in most of the tested problems, the new heuristics do not speed up the search in the normal cases. However, they do speedup the search in the bw-large-a by 60 times, and slightly improve the search time in some gridworld problems.

of the distance-based heuristics to fit the CSP context.

prob	MAX		SUM		Default	
	Normal	Skip	Normal	Skip	Normal	Skip
bw-12steps	12.98	4.43	12.79	4.43	1.96	0.66
bw-large-a	21.89	2.37	20.33	2.39	1234	1057
rocket-a	5.78	0.70	5.46	0.69	4.01	1.21
rocket-b	10.74	3.56	28.74	20.30	6.19	1.43
log-log-a	2.08	1.69	2.03	1.61	3.40	3.34
log-log-b	>1800	14.89	>1800	12.61	110	1.68
log-log-c	>1800	2.95	>1800	2.90	510	5.55
hsp-bw-02	1.11	1.11	1.19	1.19	0.89	0.89
hsp-bw-03	21.78	21.78	7.73	7.73	5.65	5.65
grid-01	25.67	25.67	24.01	24.01	36.70	36.70
grid-02	20.06	20.06	19.01	19.01	23.58	23.58

Table 8
Using hsp-values for variable and value orderings in CSP search. "Skip" means we skip the nonsolution bearing levels, and start solving from the first solution-bearing level. "Normal" means we do the CSP search in Graphplan fashion, by starting from the first level that all the goals appear non-mutex with each other. "Default" is the normal DCL variable ordering used in the GAC-CBJ solver. No EBL learning is used in the experiments. Running times are in CPU seconds.

The results for the *skip* runs, in which we start from the first solution-bearing level, are more promising. In this case, in addition to the speedup in some gridworld, and blocksworld problems, we also get improvements in 3 of 5 logistics problems. This result agrees with the observation in [20]. The contrast between the results of the normal and skip cases in the logistics problems suggests that while being fairly good in the solution bearing levels, the hsp-heuristics still spend much time doing exhaustive search in the non-solution levels, contributing to the high total-searching time in the default case. The other observation from Table 8 is that there is not much difference in performance of the *max* and *sum* heuristics. Even though there are big differences in two problems rocket-b and hsp-bw-03, the running times are very close for all remaining problems. This result shows that the value orderings are not very important compared with the variable ordering, even when we start at solution-bearing levels or higher.

7.3.3 Automatic Selection of Heuristics for Variable Ordering

Table 7 in Section 7.3.1 shows that using different heuristics can lead to performances that differ by factors of up to 100x or more on some problems. Specifically, we realized that the level-based variable ordering (LDC) tends to produce better results in the domains where the solutions are serial like *serial blocksworld* or *grid*, while the dynamic variable ordering (DLC) is better in the parallel domains such as *rocket* or *logistics*. The intuition being that in the serial domains, solutions composed of actions in consecutive levels are highly

prob	heu	level	total_f	total_a	f_mutex	a_mutex	f_ratio	a_ratio
bw-12steps	-	12	502	1100	3411	39744	**6.97**	**36.13**
bw-large-a	ldc+	12	856	2020	9575	133812	**11.18**	**66.24**
bw-large-b	ldc++	18	1996	5034	27299	505602	**13.68**	**100.44**
rocket-a	dlc++	7	370	1175	943	35999	**2.55**	**30.64**
rocket-b	dlc++	7	386	1235	1123	40454	**2.91**	**32.76**
log-a	dlc++	11	798	2074	2519	39363	**3.16**	**18.98**
log-b	dlc++	13	897	2466	3396	55735	**3.79**	**22.60**
log-c	dlc++	13	1131	3174	4696	83273	**4.15**	**26.24**
log-d	dlc++	14	2180	6442	15341	275277	**7.04**	**42.73**
hsp-bw-01	-	3	66	144	232	3451	**3.52**	**23.96**
hsp-bw-02	-	4	238	1055	1819	147697	**7.64**	**134.00**
hsp-bw-03	-	5	439	2806	4281	788519	**9.75**	**281.01**
hsp-bw-04	ldc	6	1045	10055	13903	5981802	**13.30**	**594.91**
grid-01	ldc	14	3180	7389	26101	625399	**8.21**	**84.64**
grid-02	ldc	13	2853	5952	18909	361840	**6.63**	**60.79**
grid-03	ldc+	15	3527	9191	34528	1064963	**9.79**	**115.87**
gripper-02	ldc	7	188	372	303	4085	**1.61**	**10.98**
gripper-03	ldc+	11	406	876	851	14485	**2.10**	**16.54**
hanoi-3	-	7	193	353	223	4318	**1.16**	**12.23**
hanoi-4	ldc+	15	607	1385	938	32556	**1.55**	**23.51**
hanoi-5	ldc++	31	1728	4663	3350	175174	**1.93**	**37.57**
bulldozer-1	-	9	327	647	518	11934	**1.58**	**18.45**
bulldozer-2	-	9	456	733	1137	14196	**2.49**	**19.37**
bulldozer-3	-	5	171	275	257	3070	**1.50**	**11.16**
mprime-01	-	5	443	1407	779	90396	**1.76**	**6.42**
mprime-02	ldc+	5	1245	6979	7287	1011644	**5.85**	**144.96**
mprime-04	ldc	7	836	1637	4359	63692	**5.21**	**38.91**
mystery-02	ldc+	5	1236	6790	7501	1000456	**6.08**	**147.34**
mystery-03	-	4	638	1500	2442	43467	**3.83**	**28.98**
mystery-11	ldc+	7	776	1628	4484	84417	**5.78**	**51.85**
mystery-26	ldc	6	1051	3346	3995	186967	**3.80**	**55.88**
mystery-28	ldc+	7	640	1260	2312	53760	**3.61**	**42.67**
mystery-30	ldc+	6	1518	6789	7694	901039	**5.07**	**132.72**
fridge-typed-1	-	3	140	275	38	5712	**0.27**	**20.77**
fridge-typed-2	dlc	6	287	671	89	17601	**0.31**	**26.23**

Table 9
Qualitative performance comparison for two different CSP heuristics and their relation with mutex statistics in the planning problems. "Heu": fastest heuristic; "+": 10 times or more faster than the other heuristic; "++": 50 times or more faster; "-": no difference; "level": planning graph solution level; "total_f", "total_a", 'f_mutex", "a_mutex": total numbers of facts, actions, fact mutexes, and action mutexes in the planning graph; "f_ratio", "a_ratio": ratios of f_mutex/total_f and a_mutex/total_a.

dependent on each other in the causal-effect sense. Therefore, it may be better to go level by level from the goal state. While we know of no efficient approach to detect whether the problem will have a serial or parallel solution by looking at the domain and problem specification, we suspected that there might be some approximate means of gaging the parallelism of any plan produced. Such information would have a direct bearing on the different heuristics employed. We hypothesized that for serial domains, there will tend to be more mutex constraints in a given plan graph level, effectively restricting the assignment of actions in the same level. This will ultimately lead to the lower level of parallelism.

To validate this hypothesis, we analyzed the ratios between the number of mutexes and nodes in the planning graph for various problems and domains. The eleven domains that we tested come from the AIPS-98 planning competition[27] and the distributions of various planners. We sought to test problems that are solvable and fairly represent the domain. "Fairly" here means that we tested with both small and big problems. In many domains, the biggest tested problem is one that approached the available memory limit of the Linux machine that we ran the experiments on. The eleven domains are: *serial-blocksworld*, *logistics*, *bulldozer* [14] , and *fridge-typed* from Blackbox's example domains; *parallel-blocksworld* (hsp-bw), and *gripper* from the HSP planner distribution; and *grid*, *mystery*, *mprime* from the AIPS-98 planning competition. There are other domains such as *movie*, *tire-world*, or *scheduling* (from AIPS-00 competition) that use the PDDL representation version that our planner and the blackbox-ver3.4 can not parse. All problems are tested on a P-III 500 MHz Linux machine with 256MB RAM and 512MB swap space.

Table 9 shows the statistics on the number of nodes and mutexes in the tested problems. The first column provides an indicator as to whether the DLC or LDC heuristic is superior for each problem. In this column, "-" means there is no clear performance distinction. The "+" suffix after a heuristic means that performance improves about 10 times over the other, and "++" means that the difference is about 50x or higher. For example, "ldc" means that the LDC heuristic is slightly better than DLC, "ldc+" indicates that LDC is about 10 times faster, and "ldc++" means that it is 50 times or more faster than DLC. The next column labeled "level" shows the planning graph level in which the solution is extracted. The next four columns show the total number of *facts*, *actions*, *fact-mutexes*, and *action-mutexes* in the planning graph when a solution can be found. The last two columns show two ratios: the number of fact-mutexes divided by the number of facts (f_ratio), and the number of action-mutexes divided by the number of actions (a_ratio). The results show that in the serial and parallel blocksworld, grid, mprime, and mystery domains the higher the f_ratio and a_ratio are, the better the LDC heuristic

[14] In some distributions, *bulldozer* domain is called *travel*

performed compared with DLC. Specifically, for problems with $f_ratio > 10$ or $a_ratio > 100$, LDC is generally much better than DLC. Notable exceptions are the Hanoi and Gripper domains where even though LDC is doing better, the values of a_ratio and f_ratio are very small. There is no clear explanation for those two cases, but their common characteristic is that the goals appear non-mutex in a very early level so that planners based on the building of the planning graph like Graphplan, *GP-CSP*, and Blackbox will spend a lot of time searching in many non-solution bearing levels.

Given the observation above, we employed a strategy of automatically selecting the preferred heuristic as follows:

- If $(a_ratio > 100)$ or $(f_ratio > 10)$, we classify the problem as a tightly constrained planning CSP, and we choose LDC over DLC.
- If $(f_ratio < 2)$ or $(10 f_ratio + a_ratio) < 40$, we consider the problem as loosely constrained, such as the cases of Hanoi and Gripper domains, we also choose LDC.
- For the remaining problems that have f_ratio and a_ratio values in the boundary region, LDC tends to perform slightly better in a more tightly constrained problem. We select LDC for the ones with the values $(f_ratio > 5)$ and $(a_ratio > 50)$.
- The rest are solved with the DLC heuristic.

We embedded the heuristic selection strategy described above into the *GP-CSP* planner as the default setting. The performance of our final version of *GP-CSP* with various techniques described in the previous sections is discussed in the next section.

8 Performance of the final *GP-CSP* Planner

In this section, we will discuss the performance of the final *GP-CSP* planner that includes all the improvements discussed in Sections 6 and 7. We also compare it with Graphplan and Blackbox using Satz and Relsat solvers. The final *GP-CSP* uses the compact encoding (Section 6), relevance-based EBL (Section 7.1), and the automatic heuristic selection technique (Section 7.3.3). We tested with 11 well-known domains that are collected from the planning competitions and the distributions of different planners. The origin of all domains is described in Section 7.3.3. For each domain, we tried to test with problems of various sizes, both small and large, within the memory and time limit. The problems in the *grid*, *mystery*, *mprime* and *parallel-blocksworld* domains generally require a large amount of memory. Therefore, the biggest problems for those domains in this test suite are the largest we can test in terms of our

	GPCSP		Graphplan	Satz	Relsat	speedup		
prob	heu	time (s)	time (s)	time (s)	time (s)	Graphplan	Satz	Relsat
bw-12steps	dlc	0.63	0.17	3.96	1.60	0.27	6.29	2.54
bw-large-a	ldc	5.40	0.57	27.80	32.30	0.11	5.15	5.98
bw-large-b	ldc	661	71	> 8hrs	901.55	0.11	> 43.57	1.36
rocket-a	dlc	1.22	43.13	3.81	5.27	35.35	3.12	4.32
rocket-b	dlc	2.33	87	5.91	8.39	37.34	2.54	3.60
log-a	dlc	0.95	842	2.88	1.11	886.32	3.03	1.17
log-b	dlc	19.10	390	7.73	22.03	20.42	0.40	1.15
log-c	dlc	24.27	> 8hrs	308	77	> 1187	12.69	3.17
log-d	dlc	84	> 8hrs	15.99	199.38	> 382.86	0.19	2.37
hsp-bw-02	ldc	0.34	0.32	3.62	1.21	0.94	10.65	3.56
hsp-bw-03	ldc	1.63	2.14	> 8hrs	130.77	1.31	> 17669	80.22
hsp-bw-04	ldc	4.87	19.26	> 8hrs	1682	3.95	> 5914	345.38
grid-01	ldc	7.75	7.21	> 8hrs	22.75	0.93	> 3716	2.94
grid-02	ldc	6.36	6.30	> 8hrs	21.45	0.99	> 4528	3.37
grid-03	ldc	9.83	8.77	> 8hrs	42.68	0.89	> 2930	4.34
gripper-01	ldc	0.01	0.01	0.52	0.09	1.00	52.00	9.00
gripper-02	ldc	0.41	0.05	2.41	0.69	0.12	5.88	1.68
gripper-03	ldc	62	4.28	109.72	155.72	0.07	1.77	2.51
hanoi-tower3	ldc	0.10	0.04	1.96	0.42	0.40	19.60	4.20
hanoi-tower4	ldc	9.87	0.45	12.58	54.68	0.05	1.27	5.54
hanoi-tower5	ldc	990	47.42	> 8hrs	> 8hrs	0.05	> 29.09	> 29.09
bulldozer-1	ldc	0.10	0.08	0.80	0.19	0.80	8.00	1.90
bulldozer-2	dlc	0.11	0.09	0.61	0.19	0.82	5.55	1.73
bulldozer-3	ldc	0.03	0.03	0.50	0.10	1.00	16.67	3.33
mprime-1	ldc	0.53	0.56	1.22	0.80	1.06	2.30	1.51
mprime-2	ldc	4.07	3.91	6.08	4.90	0.96	1.49	1.20
mprime-16	ldc	3.58	3.17	6.68	4.25	0.89	1.87	1.19
mystery-2	ldc	3.91	3.81	5.35	5.66	0.97	1.37	1.45
mystery-3	dlc	0.39	0.43	0.80	0.41	1.10	2.05	1.05
mystery-26	dlc	1.19	1.09	1.76	1.12	0.92	1.48	0.94
mystery-28	dlc	9.65	0.34	2.78	2.37	0.04	0.29	0.25
mystery-30	ldc	4.81	3.42	> 8 hrs	9.28	0.71	> 5988	1.93
frid-typed-1	dlc	0.12	0.12	0.59	0.19	1.00	4.92	1.58
frid-typed-2	dlc	0.38	0.34	1.34	0.65	0.89	3.53	1.71

Table 10

Comparison of the final version *GP-CSP* planner with Graphplan and Blackbox using Satz and Relsat solvers. Times is in CPU seconds. "heu" is the heuristic automatically selected by *GP-CSP* (refer to Section 7.3.1). All experiments run on a P-III 500Mhz Linux machine with 256MB of RAM.

memory limitation [15]. Problems in the *bulldozer* (travel) and *fridge-typed* are rather small, but they are also the biggest of the collection that we obtained.

Table 10 shows the running time in seconds of *GP-CSP*, Graphplan, and Blackbox with the two solvers. The last three columns shows the speedup of *GP-CSP* compared with the other three options. Compared with Blackbox running the Satz solver, *GP-CSP* is better in 31 of 34 problems with speedups up to 17669x. While *GP-CSP* is able to solve all problems, there are 8 problems that can not be solved by Satz within the time limit of 8 hours. Moreover, for the three problems in which Satz is faster, the highest speedup is only 5x.

Compared with Blackbox running the Relsat solver, *GP-CSP* is faster in 32 of 34 problems. In the only problem (mystery-28) that Relsat displays reasonable speedup over *GP-CSP* (4x), the automatic-heuristic selection of *GP-CSP* was choosing the less effective heuristic. While the speedups of *GP-CSP* over Relsat are not as radical as they are over Satz, they still reach up to 345x.

Compared with Graphplan, across 34 problems, *GP-CSP* is better in 10 problems. Including the 6 problems of the Rocket and Logistics domains with speedups ranging from 35x to more than 1187x. Graphplan is better in 21 problems. However, among them, there are 11 problems in which the speedup of Graphplan over *GP-CSP* is less than 20%. There are 4 problems in which Graphplan is more than 10x faster than *GP-CSP*, and the highest speedup of Graphplan over *GP-CSP* is about 20x in the two problems in the Hanoi domain.

It would be interesting to compare *GP-CSP* with the ILP-based planner[36], which compiles planning problems into integer linear programming encodings. However, we were unable to do it because the code was unavailable for downloading. Although we could not make a direct comparison with this type of compilation, in [36], they compared their ILP-based planner with Blackbox in the blocksworld and logistics domain. The results show that Blackbox with Satz solver is always faster than their best ILP encoding by a factor of up to 30x in all but two of the tested problems, despite the fact that the ILP-based planner was using CPLEX, a powerful commercial ILP solver from ILOG. For the relative comparison, in logistic-c problem, Blackbox with Satz is about 17 times faster than the best ILP encoding, while our planner is about 12 times faster than Satz in the same problem. Thus, *GP-CSP* is $17 * 12 = 204$x faster than the best ILP encoding in this problem.

[15] We skip all problems that are known to be unsolvable in the tested domains.

9 Related Work

Compilation approaches have become quite popular in planning in recent years. Compilation approaches construct a bounded length disjunctive structure whose substructures subsume all valid solutions of a given length. They then concentrate on identifying a substructure that corresponds to a valid solution. To achieve this extraction, they need to address two issues:

(1) Writing down a set of constraints such that any model for those constraints will be a valid plan.
(2) Compiling those constraints down into a standard combinatorial substrate.

As discussed in [26], the answers to the first question boil down to deciding which type of proof strategy to use as the basis for checking the correctness of a plan. There are essentially three standard proof strategies-corresponding to progression, regression and causal proof. The translation of the planning graph, used by *GP-CSP*, can be seen as based on a regression proof [26,22]. Tradeoffs between encodings based on different proof strategies are investigated in [22,26]-and we believe that these tradeoffs will continue to hold even when CSP is used as the compilation substrate.

Standard answers to the second question about compilation substrates include propositional satisfiability, constraint satisfaction and integer linear programming. Compilation into different types of canonical problems offers different advantages. For example, ILP encodings can exploit linear programming relaxation, which gives a *global* view of the problem, and also can naturally be mixed with Linear Programming to support continuous variables and constraints. SAT encodings can benefit from the developments of fast SAT solvers. CSP encodings can exploit the rich theory of local consistency enforcement and implicit constraint representations. Additionally, the fact that most knowledge-based scheduling work is based on CSP models [40] may make CSP encodings more natural candidates for scenarios that require close integration of planners and schedulers.

The first successful compilation approach to planning was Kautz & Selman's SATPLAN, which used a hand-coded SAT encoding for bounded length planning problems [22]. Ernst et. al. [9] extended this idea by advocating automated construction of SAT encodings from a STRIPS-type problem specification. They also studied the tradeoffs among multiple different compilation techniques. Kautz & Selman then developed the Blackbox system [21] that automatically converts the planning graph into a SAT encoding. Others, including Bockmayer & Dimopolous [3], as well as Kautz & Walser [24] considered hand-coded integer programming encodings of planning problems.

Despite the fact that the similarities between Graphplan's planning graph and CSP as well as SAT was noticed early on [19,37], van Beek & Chen [35] were the first to consider compilation of planning problems into CSP encodings. As we mentioned earlier, their emphasis in CPLAN was on hand-generating tight encodings for individual domains, and they defend this approach by pointing out that in constraint programming, domain-modeling is taken seriously. While we appreciate the efficiency advantages of hand-coded encodings, we believe that many of the facets that make CPLAN encodings effective are ones that can be incrementally automated. *GP-CSP* is a first step in that process, as it automatically constructs a CSP encoding that is competitive with other direct and compiled approaches to solving the planning graph. In the future, we expect to improve the encodings by introducing ideas based on distances [20,4] and symmetry exploitation [10]. Indeed, Wolfman [38] surveys approaches from the existing literature that could help automatically discover some of the hand-coded knowledge used in CPLAN's encodings.

As we have seen in this paper, by using implicit representations and exploiting the richer structure of the CSP problems, automatically generated CSP encodings can outperform automatically generated SAT encodings both in terms of memory and in terms of CPU time. It should be mentioned here that the recent work on lifted SAT solvers [14] provides a way of improving the memory consumption requirements of SAT encodings. We believe however that lifting is a transformation that can be adapted to CSP encodings as well.

10 Conclusion and Future directions

We have described a Graphplan variant called *GP-CSP* that automatically converts Graphplan's planning graph into a CSP encoding and solves it using standard CSP solvers. We have also presented experimental studies comparing *GP-CSP* to standard Graphplan as well as the Blackbox family of planners that compile the planning graph into SAT problems. Our comprehensive empirical studies evaluate the tradeoffs offered by encoding simplification as well as a variety of solver optimization techniques. Notable contributions include (1) a compact CSP encoding that utilizes the implicit CSP representation to reduce memory consumption and speed up solving time, (2) incorporation of EBL into the CSP solver to reduce searching time, and (3) investigations of different CSP variable ordering heuristics and a novel approach for automatic selection of heuristics based on analyzing the planning graph structure. The empirical results clearly establish the advantages of CSP-compilation approach for planning. As shown in Table 10, *GP-CSP* with our chosen default setting is superior to Blackbox (with a variety of solvers) in terms of run time in various planning problems. In addition, our indirect comparisons with ILP-based planners lend further support to CSP as the current best substrate among

compilation approaches in classical planning. *GP-CSP* generally exhibits faster runtimes and is much less susceptible to the memory blow-up problem that besets systems that compile the planning graph into SAT encodings (*GP-CSP* consistently uses much less memory than Blackbox in all tested planning problems). *GP-CSP* also runs faster than traditional Graphplan planner in several planning domains. The URL http://rakaposhi.eas.asu.edu/gp-csp.html contains our C language implementation of the *GP-CSP* system.

We are considering two different directions for extending this work –exploring more general CSP encodings and improving the CSP solvers with planning-related enhancements. In terms of the first, we plan to investigate the use of temporal CSP (TCSP) representations [6] as the basis for the encodings in *GP-CSP*. In a TCSP representation, both actions and propositions take on time intervals as values. Such encodings not only offer clear-cut advantages in handling planning problems with metric time [31], but also provide significant further reductions in the memory requirements of *GP-CSP* even on problems involving non-metric time. Specifically, many efficient Graphplan implementations use a bi-level planning graph representation [11,30] to keep it compact. The compilation strategies used in *GP-CSP*, as well as other SAT-based compilers, such as Blackbox [21], wind up unfolding the bi-level representation, losing the compression. In contrast, by using time intervals as values, a TCSP allows us to maintain the compressed representation even after compilation.

To improve the CSP solvers with planning-specific enhancements, we are considering incorporation of automatically generated state-invariants (c.f. [10]) into the CSP encoding, as well as automatically identifying variables in the encodings that should be marked "hidden" (so the CSP solver can handle them after the visible variables are handled). Most such additions have been found to be useful in CPLAN and it is our intent to essentially automatically generate the CPLAN encodings.

Finally, since most AI-based scheduling systems use CSP encodings, *GP-CSP* provides a promising avenue for attempting a principled integration of planning and scheduling phases. We are currently exploring this avenue by integrating *GP-CSP* with a CSP-based resource scheduler [32]. We model the planning and scheduling phases as two loosely coupled CSPs that communicate with each other by exchanging failure information in terms of graphplan style abstract no-goods[16].

References

[1] R. Bayardo and D. Miranker. A Complexity Analysis of Space-Bounded Learning Algorithms for the Constraint Satisfaction Problem. In *Proc. of the*

13th Nat'l Conf. on Artificial Intelligence, 1996.

[2] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2), 1997.

[3] A. Bockmayr and Y. Dimopolous. Mixed integer programming models for planning problems. In *In CP'98 Workshop on Constraint Problem Reformulation*, 1998.

[4] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proc. 5th European Conference on Planning*, 1999.

[5] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proc. AAAI-97*, 1999.

[6] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. In *Artificial Intelligence 49*, 1991.

[7] M. Do, B. Srivastava, S. Kampahampati. Investigating the Effect of Relevance and Reachability Constraints on SAT Encodings of Planning. In *Proc. AIPS-2000*, 1997.

[8] M. Do, S. Kampahampati. Solving Planning-Graph by Compiling it into CSP. In *Proc. AIPS-2000*, 2000.

[9] M. Ernst, T. Millstein, and D. Weld. Automatic SAT compilation of planning problems. In *Proc. IJCAI-97*, 1997.

[10] M. Fox and D. Long. The detection and exploitation of symmety in planning domains. In *Proc. IJCAI-99*, 1999.

[11] M. Fox and D. Long. Efficient implementation of plan graph. *Journal of Artificial Intelligence Research*, 10, 1999.

[12] J. Frank, A. Jonsson, and P. Morris. On Reformulating Planning As Dynamic Constraint Satisfaction (Extended Abstract) *Symposium on Abstraction, Reformulation and Approximation*, 2000.

[13] D. Frost and R. Dechter. Dead-end driven learning In *Proc. AAAI-94*, 10, 1999.

[14] M. Ginsberg and A. Parkes. Satisfiability algorithms and finite quantification. In *Proc. KR-2000*, 2000.

[15] S. Kambhampati. Challenges in bridging plan synthesis paradigms. In *Proc. IJCAI-97*, 1997.

[16] S. Kambhampati. Improving graphplan's search with ebl & ddb techniques. In *Proc. IJCAI-99*, 1999.

[17] S. Kambhampati. On the relation between intelligent backtracking and failure-driven explanation-based learning in constraint satisfaction and planning In *Artificial Intelligence*, Spring 1999.

[18] S. Kambhampati. Planning Graph as a (dynamic) CSP: Exploiting EBL, DDB and other CSP search techniques in Graphplan. *Journal of Artificial Intelligence Research*, 2000.

[19] S. Kambhampati, E. Parker, and E. Lambrecht. Understanding and extending graphplan. In *Proc. of 4th European Conference on Planning*, 1997. URL: rakaposhi.eas.asu.edu/ewsp-graphplan.ps.

[20] S. Kambhampati and R. Sanchez. Distance-based goal-ordering techniques for graphplan. In *Proc. AIPS-00*, 2000.

[21] H. Kautz and B. Selman. Blackbox: Unifying SAT-based and graph-based planning. In *Proc. IJCAI-99*, 1999.

[22] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proc. AAAI-96*, 1996.

[23] H. Kautz, D. McAllester and B. Selman. Encoding Plans in Propositional Logic In *Proc. KR-96*, 1996.

[24] H. Kautz and J. Walser. State-space planning by integer optimization. In *Proc. AAAI-99*, 1999.

[25] J. Hoffman. A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-climbing Algorithm. *Technical Report*, Institut fr Informatik, 2000.

[26] A. Mali and S. Kambhampati. On the utility of Plan-space (causal) Encodings. In *Proc. AAAI-99*, 1999.

[27] D. McDermott. AIPS-98 Planning Competition Results. At ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html, 1998.

[28] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proc. AAAI-90*, 1990.

[29] J. Rintanen. A planning algorithm non-based on directional search. In *Proc. KR-98*, 1998.

[30] D. Smith and D. Weld. Temporal planning with mutual exclusion reasoning. In *Proc. IJCAI-99*, 1999.

[31] D. Smith, J. Frank, and A. Jonsson. Bridging the gap between Planning and Scheduling. In *Knowledge Engineering Review 15:1*, 2000.

[32] B. Srivastava, S. Kambhampati, and M. Do. Planning the Project Management Way: Efficient Planning by Effective Integration of Causal and Resource Reasoning in RealPlan. *To appear in Aritificial Intelligence Journal*, 2001.

[33] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, San Diego, California, 1993.

[34] P. van Beek. *CSPLIB: A library of CSP routines.* University of Alberta, http://www.cs.ualberta.ca/ vanbeek, 1994.

[35] P. van Beek and X. Chen. CPlan: A constraint programming approach to planning. In *Proc. AAAI-99*, 1999.

[36] T. Vossen, M. Ball, A. Lotem, and D. Nau. On the Use of Integer Programming Models in AI Planning. In *Proc. IJCAI-99*, 1999.

[37] D. Weld, C. Anderson, and D. Smith. Extending graphplan to handle uncertainty & sensing actions. In *Proc. AAAI-98*, 1998.

[38] S. Wolfman. Automatic discovery and exploitation of domain knowledge in planning. Generals Paper. University of Washington, 1999.

[39] T. Zimmerman and S. Kambhampati. Exploiting symmetry in the plan-graph via explanation-guided search. In *Proc. AAAI-99*, 1999.

[40] M. Zweben and M. Fox. *Intelligent Scheduling*. Morgan Kaufmann, 1994.