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

Minh Binh Do*& Subbarao Kambhampati
Department of Computer Science and Engineering
Arizona State University, Tempe AZ 85287-5406
Email: {binhminh,rao}@asu.edu

June 30, 2000

Abstract

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. This paper describes GP-CSP, a system that does planning by automatically converting Graphplan's planning graph into a CSP encoding, and solving the CSP encoding using standard CSP solvers. Our comprehensive empirical evaluation of GP-CSP demonstrates that it is superior to both standard Graphplan and Blackbox system, which compiles planning graphs into SAT encodings. Our results show that CSP encodings outperform SAT encodings in terms of both space and time requirements. The space reduction is particularly important as it makes GP-CSP less susceptible to the memory blow-up associated with SAT compilation methods. Our work is inspired by the success of van Beek & Chen's CPLAN system. However, in contrast to CPLAN, which expects *hand-coded* CSP encodings for individual domains and problems, GP-CSP is able to take domain descriptions in STRIPS (PDDL) representation, and automatically generate the CSP encodings.

1 Introduction

Since the development of the original Graphplan algorithm [2], several researchers [18, 33] have noted the striking similarities between the backward search phase of Graphplan, and constraint satisfaction problems [30]. In most cases however, the detection of similarities has lead to adaptation of CSP techniques to Graphplan. For example, our own recent work [14, 16] has considered the utility of adapting the explanation-based learning and dependency directed backtracking strategies from CSP to backward search phase of Graphplan. More recently, researchers from CSP have started taking interest in applying constraint programming to classical planning. van Beek & Chen [32] 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

*We thank Biplav Srivastava for explaining the inner workings of van Beek and Chen's constraint solver, and for 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, Army AASERT grant DAAH04-96-1-0247, 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

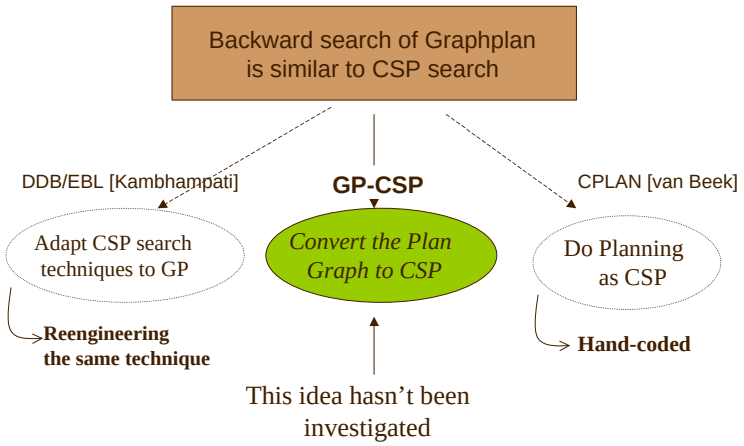


Figure 1: GP-CSP framework in comparison with other efforts to take advantage of the similarity between Graphplan's backward search and the CSP search.

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 that solves the planning graphs by automatically converting them into CSP encodings. Figure 1 shows how GP-CSP differentiates itself from other frameworks that make use of the similarity between the backward search of Graphplan and the CSP search. 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[31]. Our empirical studies show that GP-CSP is significantly superior to Graphplan as well as Blackbox which compiles planning problems into SAT encodings. While GP-CSP's dominance over standard Graphplan 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 [26] 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 Blackbox [21] are typically highly susceptible to memory blow-up[1]. CSP encodings used by GP-CSP are much less susceptible to this problem for two reasons. In general, SAT encoding of a problem tends

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

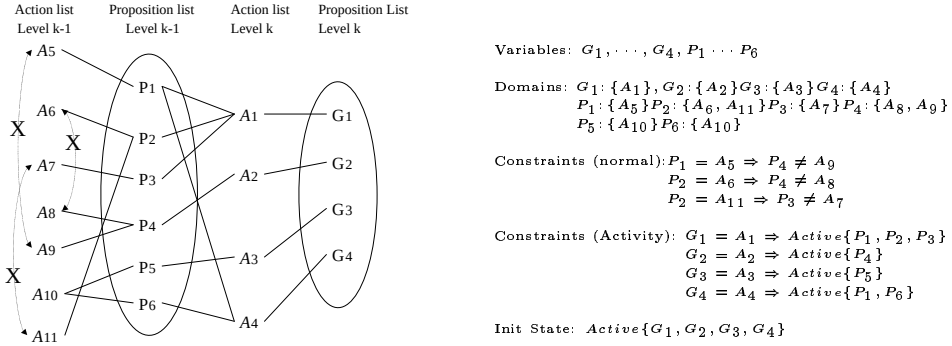
to be larger than the CSP encodings used in GP-CSP (at least in terms of variables). Second, GP-CSP is able to use implicitly specified constraints (c.f. [31]). This could keep the size of the encoding down considerably.

- 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 improved structure can be exploited in developing directed partial consistency enforcement 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 problems [36]. Approaches like GP-CSP may thus provide better substrates for integrating planning and scheduling. In fact, in related work [29], we discuss how CSP techniques can be used to tease resource scheduling away from planning.

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 planning graph can be automatically encoded into a (dynamic) CSP problem. In Section 4, we describe the way GP-CSP automatically converts planning graph into a CSP encoding in a format that is handled by a the CSP library developed by van Beek[31]. Section 5 describe 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 improvements to the CSP solver in Section 7. Section 8 discusses the relation to other work and Section 9 summarizes the contributions of the paper and sketches several directions for future work.

2 Review of Graphplan algorithm

Graphplan algorithm [2] can be seen as a "disjunctive" version of the forward state space planners [18, 13]. 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 2 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 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. Planning-graph maintains the dependency links between the actions at 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, with the actions that are statically interfering with each other (i.e., their preconditions and effects are inconsistent) labeled



(a) Planning Graph (b) DCSP

Figure 2: A planning graph and the DCSP corresponding to it

mutex. Mutexes are then propagated from this level forward by using a 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 planning-grap is grown 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 2 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. 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 $P_1 \cdots P_6$.)

3 Connections between Graphplan and CSP

The process of searching the planning graph to extract a valid plan from it can be seen as a dynamic constraint satisfaction problem. The dynamic constraint satisfaction problem (DCSP) [25] (also sometimes called a conditional CSP problem) is a generalization of the constraint satisfaction problem [30], that is specified by a set of variables, activity flags for the variables, the domains of the variables, and the constraints on the legal variable-value combinations. In a DCSP, initially only a subset of the variables is active, and the objective is to *find assignments for all active variables that is 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[2], 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:

$$\neg (p_{11} = a_1 \wedge p_{12} = a_2)$$

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

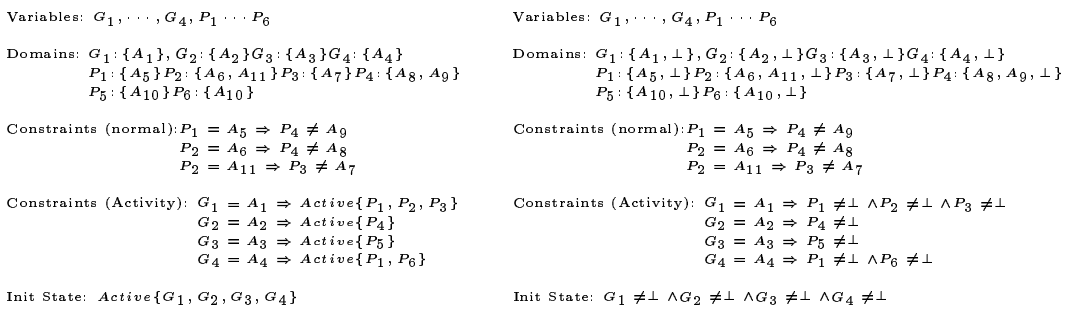
$$\neg (Active(p_{11}) \wedge Active(p_{12}))$$

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 2 shows the dynamic constraint satisfaction problem corresponding to the example planning-graph that we discussed.

There are two ways of solving a DCSP problem. The first, direct, approach [25] 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

[2]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 2 contains syntactically different literals in different levels of the graph.



(a) DCSP (b) CSP

Figure 3: Compiling a DCSP to a standard CSP

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 the previous epoch and try to find an alternative satisfying assignment to those variables (backtracking further, if no other assignment is possible). 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.

The second approach for solving a DCSP is to first 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 about ensuring that propositions that are preconditions of a selected action do take 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.

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 [30]. [3]

It is also worth noting [18], most of the mutex constraints are "derived" constraints and are thus redundant. Soundness is guaranteed as long as we keep mutex constraints corresponding to static interferences between actions. The remaining propagated action mutexes, as well as all fact mutex constraints are redundant.

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 plan 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 supports 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.[4]

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 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 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.

[3] 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].

[4] Compilation to CSP is not a strict requirement for doing non-directional search. In [35], we describe a technique that allows the backward search of Graphplan to be non-directional, see the discussion in Section 9.

In the default SAT encoding of Blackbox, 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) [5].

The increase is necessary because in CSP, actions are not variables, and that mutual exclusions between actions has to be modeled indirectly as constraints on legal variable-value combinations. In Section 6, we describe how we can exploit the implicit nature of constraints in GP-CSP to reduce the constraints.

The fact that direct translation of planning graph into CSP leads to 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. Nevertheless, there may be domains where the savings in memory through implicit representation does not offset the increase due to the number of constraints.

4 Implementation details of Compiling Planning Graph to CSP

As mentioned in the previous section, GP-CSP uses the CSP encoding of the planning graph. The basic idea is to let Graphplan build the planning graph representation, and convert into a CSP encoding, along the lines illustrated in Figure 3. We use the CSP library created by van Beek[31], 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 is to reduce 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.

[5] Notice that all of fact mutex, 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 resulted from doing directional partial 2-consistency in graph expansion phase.

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 is placed as the first value in the domain of each variable.

4. **Setting up the constraints:** van Beek's CSP library allows for definition of implicit constraints. It does this by allowing 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 constraint*, *fact mutex constraint* and *action mutex constraint*. 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 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 the Initial Encoding

We have implemented GP-CSP completely, and have begun comparing 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, the differences in performance between those three are solely between their searching time, and conversion time. As the matter of fact, the time to convert the plangraph to CNF

	GP-CSP			Graphplan			Satz			Relsat		
	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
log-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
log-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
frid02	0.97	1 M	13/6	0.65	1 M	13/6	3.03	62 M	13/6	1.40	67 M	13/6
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 ran in Sun Ultra 5, 256 M RAM machine. Hsp-bw-04 was ran on a Pentium 500 MHz machine running LINUX with 512 M RAM.

form in Blackbox, and to the CSP encoding in GP-CSP are similar, and are quite trivial compared with graph expansion, and searching times. For example, in problem log-b, Blackbox spends 0.12s for converting a grahph, 1.1s for expanding, and around 16.7s for searching. For the same problem, our best GP-CSP implementation takes 0.11s for conversion, 1.1s for expanding the graph, and 2.79s for solving the CSP encoding [6] . 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 CPLAN system used [32]. Table 1 compares the performance of these systems on a set of benchmark problems taken from the literature. The results show that GP-CSP is competitive with Graphplan as well as Blackbox with two state-of-the-art solvers–SATZ and Relsat [7] . 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 logistic, in which GP-CSP and two SAT solvers are quite close 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 CSP encoding uses to represent 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 an higher number of mutex constraints in GP-CSP formulation, compared with SAT. Nevertheless, GP-CSP was still able to outperform bot SATZ and Relsat in that domain.

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. However, for all but one problem, the standard directional backward search of Graphplan returns shorter solutions. 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 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

[6]Note that we did not mention the time each SAT solver in Blackbox need to convert the CNF form to their own structure. This extra time is not needed in our GP-CSP system, because we convert directly from the plangraph to the structure that the GAC-CBJ solver can use.

[7]To make comparisons meaningful, we have run the SATZ and Relsat solvers without the random-restart strategy, and setting the cutoff-limit to 1000000000. This is mainly because random-restart is a technique that is not unique to SAT solvers; see for example [16] for the discussion of how random-restart strategy was introduced into Graphplan's backward search. However, the running times of sat solvers are still depended on the initial random seeds, so we take an average of 10 runs for each problem.

	FC	AC	Only A-Mutex (with FC)
bw-12steps	7.59	33.37	7.54
bw-large-a	138	1162	140
log-rocket-a	9.25	23.14	8.31
log-rocket-b	19.42	49.34	19.48
log-a	16.19	32.73	83
frid02	0.97	1.57	1.05
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.

strategy considers $\perp$ value for every variable (proposition), as long as it is not forbidden to consider that value. We are currently considering adding constraints to capture the *only if* part of the activation, to see if that improves quality without significantly worsening performance.

We also did some preliminary experimentation 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. 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. Forward checking involves doing constraint propagation only when all but one of the variables of a constraint are instantiated. Arc-consistency is more eager and attempts propagation even if two (i.e., all–since we only have binary constraints) of the variables in the constraint are un-instantiated. 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 decided to keep them in the default encoding of GP-CSP encoding.

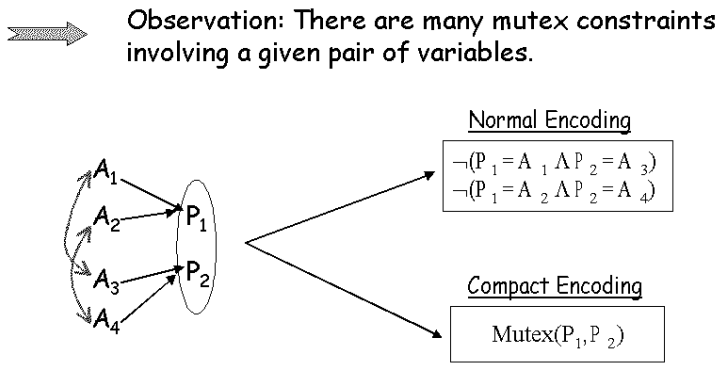


Figure 4: Illustration of the differences between the Explicit and Implicit representation of mutex constraints.

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 $O(e^2)$ constrains on the proposition-action (variable-value) combinations– leading to a total of $O(ln^2e^2)$ constraints. While explicit representation of mutex constraints allowed GP-CSP to win over SAT encodings, in terms of memory, in most cases, the increased number of constraints do increase its memory consumption, especially in domains such as the parallel (HSP) blocks world.

We have devised a method that uses the 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 number of proposition and actions per level, and l is the length of the planning graph), 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, and following are the details of this "compact" encoding:

- 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 relation between supporting actions in the compact encoding. In order to check the

	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
log-rocket-a	9.25	5 M	26/7	21921	23147	4.01	3 M	26/7	4992	6218
log-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
frid02	0.97	1 M	13/6	14326	17696	0.83	1.2 M	13/6	222	3592
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

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 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, the number of constraints needed to model action mutexes is $O(lm^2)$–since in the worst case, every pair of m 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 constraint 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 reduced by 4-140 times in the compact encoding, compared with the normal one. As the result, the memory consumed by GP-CSP, which is shown in the "mem" columns of table 3, is reduced from 4-6 times for problems that use more than 10 MB of memory, and is now always less than that consumed by Blackbox (see Table 1). The new encoding also seems to be easier to solve in all but one problem. In particular, problem log-b and hsp-bw-04 can be solved 28 and 20 times faster than the normal encoding. For most of the other problems we also get speedup of up to 4x. The only problem that experiences considerable slowdown is bw-large-a, which is an easy problem to begin with. Thus, compact encoding is superior to

the direct encoding.

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 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 investigated by us include: (1) explanation based learning (EBL) (2) level-based variable ordering, (3) random restart search with cutoff limit on backtracks (4) distance based variable and value ordering [17], (5) min-conflict value ordering, and (6) the use of bmutex constraints [23]. In our experiments to-date, only the first four enhancements have demonstrated significant improvements in performance. We thus limit our discussion to these four enhancements. Complete details of our experiments can be found in [7].

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 use those explanations to avoid similar failures later [15]. 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 [11].

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 Two of the best-known solutions for this problem in CSP are size-based learning [11], and relevance-based learning [1]. A k-degree size-based learning will ignore 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 ignores 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 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 requirement 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. Especially for the parallel blocksworld domain, significant speedups only occur with $k \geq 30$.

Problem	size-based EBL						relevance-based EBL				Speedup /Mem-ratio for GP-CSP + Rel-10 EBL			
	3-t	3-m	10-t	10-m	30-t	30-m	10-t	10-m	5-t	5-m	GP-CSP	Graphplan	SATZ	Relsat
bw12steps	1.69	11M	**1.31**	11M	1.50	11	1.46	10M	1.40	10	1.34/0.30	0.28/0.1	5.60/6.40	2.10/7.00
bw-large-a	608	24M	259	24M	173	26M	134	26M	**128**	24M	9.21/0.42	0.01/0.12	0.36/3.38	0.22/3.34
rocket-a	3.69	8M	**2.08**	8M	2.49	8M	2.39	11M	2.59	11M	1.68/0.27	28.45/5.54	3.72/6.36	3.76/6.63
rocket-b	5.52	9M	**3.55**	9M	4.33	9M	4.00	10M	4.31	10M	1.55/0.4	32.5/9.50	2.94/7.00	4.47/7.10
log-a	2.67	15M	2.37	18M	**2.26**	18M	2.30	18M	2.60	18M	1.45/0.22	770/9.83	3.07/4.00	1.91/4.22
log-b	59.58	18M	39.55	19M	48.22	29M	**35.13**	19M	36.77	18M	3.13/0.29	22.40/4.21	0.46/4.16	1.32/4.21
log-c	153	24M	61	24M	63	34M	**48.72**	25M	65	25M	10.47/0.88	>220	24.42/3.36	2.61/3.56
hsp-bw02	1.08	12M	**1.03**	12M	1.14	12M	1.09	12M	1.05	12M	0.82/0.37	0.79/0.08	6.56/5.67	2.27/5.50
hsp-bw03	5.08	26M	**5.04**	26M	5.19	26M	5.17	41M	5.12	41M	0.86/0.32	0.98/0.59	>5570	37.52/2.95
hsp-bw04*	40.41	86M	38.01	86M	24.07	89M	**23.89**	86M	26.57	86M	1.65/0.75	0.81/0.97	>1205	70.41/1.79
grid-01	23.26	68M	21.60	68M	**21.46**	68M	21.98	62M	22.28	57M	1.07/1.00	0.82/0.66	>491	1.45/1.81
grid-02	36.78	92M	35.77	92M	**25.93**	92M	26.69	92M	25.99	88M	1.38/0.89	0.85/0.67	> 405	2.48/1.5
grid-03	124	127M	108	127M	58.68	127M	53.68	128M	**53.51**	128	2.38/0.88	0.51/0.53	> 201	1.83/1.33

Table 4: Incorporating EBL into GAC-CBJ. Times are in second. All problems are ran in Sun Ultra 5 Unix machine with 256 MB of memory. To be consistent with other tables, problem hsp-bw04 is ran in Linux machine.

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

Table 5: GP-CSP with different variable orderings. The EBL used in this experiment size-based EBL with maximum nogood size is set to 10. All experiments are done in the Ultra5 Unix machine, except hsp-bw4, which is ran in Linux 500MHz machine.

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 four 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 consumption. We see that relevance-based learning is generally faster than the best size-based learning on 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 last 4 columns in table 4 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), 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 is still consistently 2 to 7 times smaller than Blackbox using both SATZ, and Relsat solvers. GP-CSP is faster than Blackbox with 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[8].

[8]The solutions returned by GAC-CBJ-EBL is the same with the ones returned by GAC-CBJ

	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
log-rocket-a	3.69	8MB	**2.08**	8MB	2.49	8MB	3.75	8MB	**2.57**	8MB	2.63	8MB
log-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 the values of k are 3,10, and 30. For each experiment, the two columns show the time in seconds, and memory consumptions in MB.

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 own backward 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 however 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 encoding 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 planning graph ensures that every variable in k^{th} level encoding is also present, with identical domain and inter-variable constraints, in the $k+1^{th}$ level encoding. This 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 $N : x_1 = v_1 \cdots x_i = v_i$ at k^{th} level that

was produced only because G_k was required to have 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 $x_1 = v_1 \cdots x_i = v_i \wedge G_k \neq \perp$, 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 specific failures, and conjoining them to the learned nogoods. In [15, Section 4.1], Kambhampati provides straighforward algorithms for generating such contextualized nogoods, and we adapted those algorithms for GP-CSP.[9]

Although we managed to implement this inter-level nogood usage and verify its correctness, we found, to our disappointment, that reusing recorded nogoods does not after all provide a favorable cost-benifit ratio. We found that use of such inter-level nogoods lead to consistently worsened performance than using intra-level nogoods alone in most of the problem. Table 6 shows the comparison results between 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 didn't 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 [16]). In particular, as pointed out in [16], Graphplan's memos can be seen as nogoods of the form $P_1 \neq \perp \wedge \cdots \wedge P_j \neq \perp$ 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 from the previous encodings seems to decrease, driving down the utility of storing and matching the previous level nogoods.

It is of course possible to increase the reusability of nogoods by concentrating only on 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).[10]. 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.

[9] 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.

[10] Notice that if we do not go from level by level, then 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.

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, will also help GP-CSP to speed up the search in those domains. Currently, the GAC-CBJ solver used in GP-CSP uses the *dynamic variable ordering* which prefers variables with smaller live domains (D), breaks ties by the *most-constrained variable ordering* which prefers variables that take part in more constraints(C), followed by the level-based variable ordering (L) which prefers variables from higher levels of the planning graph. Let us call this default strategy the DCL strategy. DCL strategy gives tertiary importance to the level information. To make variable ordering more Graphplan-like, we tried two other variable orderings LDC, which gives primary importance to level, and DLC which gives it secondary importance. The peformance of these three variable ordering strategies are compared in Table 7.1. As we can easily see, the new variable orderings significantly speedup the GP-CSP in the two blocksworld domains, but slow the search down in the logistics domain.

7.3.2 HSP-based Variable and Value Ordering

This 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 the variation of the variable and value ordering[11] heuristics used in the Graphplan algorithm that is discussed in [17]. These heuristics are based on the *difficulty* of achieving propositions in the 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 plan graph structure. The difficulty of 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, 17]. Specifically, we compute the hsp values for all fact and action nodes in the graph as follows:

- The hsp value of each fact will be the value of the first level in which it appears in the plan graph.
- The hsp value of each action will be 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. Using the

[11] Because the backward search of graphplan go backward level by level, and choose one set of actions at a time, the heuristics discussed in [17] 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 way to calculate, and use of the distance-based heuristics to fit in the CSP context.

prob	MAX		SUM		Default	
	Skip	Normal	Skip	Normal	Skip	Normal
log-rocket-a	0.70	5.78	0.69	5.46	1.21	4.01
log-rocket-b	3.56	10.74	20.30	28.74	1.43	6.19
log-log-a	1.69	2.08	1.61	2.03	3.40	3.34
log-log-b	14.89	>1800	12.61	>1800	1.68	110
log-log-c	2.95	>1800	2.90	>1800	5.55	510
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
bw-12steps	4.43	12.98	4.43	12.79	0.66	1.96
bw-large-a	2.37	21.89	2.39	20.33	1057	1234
grid-01	25.67	25.67	24.01	24.01	36.70	36.70
grid-011	20.06	20.06	19.01	19.01	23.58	23.58

Table 7: Using HSP values to 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 the normal way like Graphplan, by starting from the first level that all the goals appear non-mutex with each other.

hsp values to guide the search still follows the CSP's basic strategy of choosing the most difficult (in assigning value) variable first, and the least constrained value first. However, the difficulty here is not measured by the number of remaining value (DVO), or the number of constraint that a variable participate 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 the selected variable, choosing the value in its domain corresponding to the action with the smallest hsp value.

Table 7 shows the results for using the HSP values for variable and value orderings for a set of benchmark problems. Values in the table are running time in seconds. The variable orderings in both of the experiments are based on hsp values as described above. 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. As suggested by [17], we tested with two cases for each problem: normal case, in which we start searching from the level that all the goals 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.

The results for the second testing case, in which we start from the first solution-bearing level, are more promising. In this case, besides the speedup in some of the gridworld, and

prob	GP-CSP	SATZ	Relsat
rocket-a	1.79	7.98	8.33
rocket-b	1.90	11.62	15.57
log-a	2.37	6.27	3.73
log-b	3.64	19.30	42.21
log-c	6.57	46.27	62

Table 8: GP-CSP with cutoff limit of 5000

blocksworld problems, we also get the improvements in 3 of 5 logistics problems. This result agrees with the observation in [17]. The contrast of the results of the normal and skip cases in the logistics problems suggests that while being fairly good in the solution bearing level, the hsp-heuristics have been misleading the search in the non-solution levels, which contributes to the high total-searching time in the default case. The other observation from table 7 is that there is not much difference in performance between the *max* and *sum* heuristics. Even though there are big differences in two problems log-rocket-b, and hsp-bw-03, the running times are very close for all the remaining problems. This result shows that the value orderings are not very important, compared with the variable ordering, even in the solution-bearing search.

7.4 Random-restart search

We also tested the strategy of randomly restarting the search after a limited number of backtracks. Such a strategy has been found to be quite effective in handling heavy-tail distribution of solutions in many planning domains. Table 8 shows the result of setting the cutoff-limit value of 5000 for a class of logistic problems. The table shows that the speedups from random restart search do not change the relative superiority of GP-CSP over Blackbox. With the cutoff limit of 5000, GP-CSP is up to 10x better than both sat solvers in all problems. However, setting the right cutoff limit for different problems in different domains is very tricky issue that has not received enough attention.

8 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 do 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 [24], the answers to the first question boil down to deciding which type of proof strategy to use as the basis for checking 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 [24, 20]. Tradeoffs between encodings based on different proof strategies are investigated in [20, 24]–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, IP encodings can exploit the linear programming relaxations, which give a *global* view of the problem, and also provide natural support for continuous variables. 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 [36] 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 hand-coded SAT encodings of bounded length planning problems [19]. Ernst et. al. [8] 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 [22] 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 were noticed early on [18, 33], van Beek & Chen [32] 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 planning graphs. In future, we expect to improve the encodings by introducing ideas based on distances [17, 4], and symmetry exploitation [9]. Indeed Wolfman [34] surveys approaches from 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 [12] provides a way of improving the memory consumption requirements of SAT encodings.

We believe however that lifting is a transformation that can also be adapted to CSP encodings.

9 Conclusion and Future directions

We have described a Graphplan variant called GP-CSP that automatically converts the Graphplan's planning graph into a CSP encoding, and solves it using standard CSP solvers. We have described our experimental studies in comparing GP-CSP to standard Graphplan as well as the Blackbox family of planners that compile planning graph into a SAT problems. Our comprehensive empirical studies evaluate the tradeoffs offered by a variety of encoding simplifications as well as solver optimization techniques. The results clearly establish the advantages of CSP-compilation approaches for planning. GP-CSP is superior to both standard graphplan and Blackbox (with a variety of solvers) in terms of the time–significantly outperforming these systems on many problems. More importantly, GP-CSP is much less susceptible to the memory blow-up problem that besets the systems that compile planning graph into SAT encodings. The URL http://rakaposhi.eas.asu.edu/gp-csp.html contains our C language implementation of the GP-CSP system.

While our progress till now has been promising, in a way, we have just begun to scratch the surface in terms of exploiting the possibilities offered by CSP encodings.

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 [27], 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 [10, 28] 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. [9]) 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 [29]. 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[14].

References

[1] R. Bayardo and D. Miranker. A complexity analysis of space-bounded larning 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 *In Proc. AAAI-97*, 1999.

[6] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence 49*, 1999.

[7] M. Do and S. Kampahampati. Solving planning-graph by compiling it into csp. In *Proc. AIPS-2000*, 2000.

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

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

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

[11] D. Frost and R. Dechter. Dead-end driven learning. In *Proc. AAAI-94*, 1994.

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

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

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

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

[16] S. Kambhampati. Planning graph as a (dynamic) csp: Exploiting ebl, ddb and other csp search techniques in graphplan. *Journal of Artificial Intelligence Research*, 1999.

[17] S. Kambhampati and R.S. Nigenda. Distance-based goal ordering heuristics for graphplan. Technical Report ASU CSE TR 99-010, Arizona State University, 1999.

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

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

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

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

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

[23] S. Kampahampati M. Do, B. Srivastava. Investigating the effect of relevance and reachability constraints on sat encodings of planning. In *Proc. AIPS-2000*, 2000.

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

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

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

[27] D. Smith, J. Frank, and A. Jonsson. Bridging the gap between planning and scheduling. *Knowledge Engineering Review 15:1*, 2000.

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

[29] B. Srivastava and S. Kambhampati. Scaling up planning by teasing out resource scheduling. In *Proc. ECP-1999*, 1999.

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

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

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

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

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

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

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