On the relations between Intelligent Backtracking and Failure-driven Explanation Based Learning in Constraint Satisfaction and Planning

ASU CSE TR 97-018

Subbarao Kambhampati rao@asu.edu
Department of Computer Science and Engineering
Arizona State University, Tempe, AZ 85287.

Abstract

The ideas of intelligent backtracking (IB) and explanation based learning (EBL) have developed independently in constraint satisfaction, planning, machine learning and problem solving communities. The variety of approaches developed for IB and EBL in the various communities have hither-to been incomparable. In this paper, I formalize and unify these ideas under the task-independent framework of refinement search, which can model the search strategies used in both planning and constraint satisfaction (CSP). I show that both IB and EBL depend upon the common theory of explanation analysis– which involves explaining search failures, and regressing them to higher levels of the search tree. My comprehensive analysis shows that most of the differences between the CSP and planning approaches to EBL and DDB revolve around different solutions to: (a) How the failure explanations are computed (b) How they are contextualized (Contextualization involves deciding whether or not to keep the flaw description and the description of the violated problem constraints) and (c) How the storage of explanations is managed. The differences themselves can be understood in terms of the differences between planning and CSP problems as instantiations of refinement search. This unified understanding is expected to support a greater cross-fertilization of ideas among CSP, Planning and EBL communities.

1. Introduction

One of the main-stays of AI literature is the idea of "intelligent backtracking" as an antidote for the inefficiencies of chronological backtracking (Stallman & Sussman, 1977). However, there is a considerable confusion and variation regarding the various implementations of intelligent backtracking. Many apparently different ideas, such as back jumping, nogood-based learning and dynamic backtracking are all concerned with the general notion of intelligent backtracking. Complicating the picture further is the fact that many "speedup learning" algorithms that learn from failure (c.f. (Minton, Carbonell, Knoblock, Kuokka, Etzioni, & Gil, 1989; Bhatnagar & Mostow, 1994; Kambhampati, Katukam, & Qu, 1996; Dechter, 1990; Schiex & Verfaillie, 1993)), do analyses that are quite close to the type of analysis done in the intelligent backtracking algorithms. Although this similarity has sometimes been noted in earlier literature (c.f. (Dechter, 1990)), a thorough analysis has been impeded by the many superficial differences between the existing approaches in CSP and Planning.

My motivation in this paper is to put the different ideas and approaches related to IB and EBL in planning and CSP in a common perspective, and thereby delineate the underlying commonalities between research efforts that have so far been seen as distinct or at best losely connected. To this end, I consider all backtracking and learning algorithms within the context of general refinement search (Kambhampati, Knoblock, & Yang, 1995). Refinement search involves starting with the set of all potential solutions for the problem, and repeatedly splitting the set until a solution for the problem can be extracted from one of the sets. The common algorithms used in both planning and CSP can be modeled in terms of refinement search.

I show that within refinement search, both IB and EBL depend upon a common theory of explaining search failures, and regressing them to higher levels of the search tree to compute explanations of failures of the interior nodes. I argue that intelligent backtracking is best understood as "*explanation directed backtracking*" (EDB)[1] which occurs any time the explanation of failure regresses unchanged over a refinement decision. At that point, we can ignore all siblings of that decision, and continue backtracking to the next higher level. Most of the existing backtracking algorithms can be understood as specializations or extensions of this idea (see Section 5). EBL involves remembering the interior node failure explanations and using them in future to prune unpromising branches.

Within this framework, a multitude of variations are possible depending on how the failures are represented, contextualized, and how many of them are stored for future use. I will show how approaches for CSP and planning differ in these aspects, and justify these differences in terms of the characteristics of CSP and Planning problems, when seen as instantiations of refinement search. In addition, I will discuss how ideas such as "constraint propagation" (Tsang, 1993) and "dynamic backtracking" (Ginsberg & McAllester, 1994) are related to the ideas of IB and EBL.

The main contribution of this paper is thus pedagogical in nature– it uses a rational reconstruction of the ideas behind IB and EBL to relate and unify the hither-to disparate bodies of work in planning, CSP, and EBL. As van Harmalen and Bundy (1988) point out, such rational reconstructions of apparently unrelated algorithms and approaches in terms of each other is a useful activity, not only because it prevents reinventing the wheel, but also because often such rational reconstructions generate new insights in and additions to both areas. We will demonstrate that the unified task-independent understanding of IB and EBL helps a crisp statement of the tradeoffs offered by the different algorithms and can support cross-fertilization of ideas among the CSP, planning and EBL communities.

The insights gained from this paper may in fact be quite timely. Although work on dependency directed backtracking and EBL have been dormant in recent years, there are several reasons to expect a resurgence of interest these topics. Much of the early work in CSP has been on systematic search algorithms, within which EDB and EBL play a role. A variety of empirical studies (c.f. (Prosser, 1993; Frost & Dechter, 1994c, 1994a)) have consistently shown that EDB and EBL techniques are often part of the winning constraint satisfaction search algorithms. Although the emphasis shifted to non-systematic search strategies such as GSAT (Selman, Levesque, & Mitchel, 1992) in the recent past, there is now new evidence (c.f. (Bayardo & Schrag, 1996, 1997)) that systematic search algo-

1. I use the term EDB rather than the more common "dependency directed backtracking" since the latter has been used by some authors to refer to both intelligent backtracking and learning from failures. We shall see that these ideas are best studied separately.

rithms, armed with EDB and EBL mechanisms[2] can outperform non-systematic searchers such as GSAT and WALKSAT (Selman et al., 1992) on several hard real and artificial satisfiability instances. Similarly, within the planning and problem-solving communities, EBL approaches are finding continued uses in learning search control (Kambhampati et al., 1996), case-based planning (Ihrig & Kambhampati, 1996; Munoz-Avila & Weberskirsch, 1996), plan quality control (Estlin & Mooney, 1996). Moreover, recent work in planning has amply emphasized the role of constraint satisfaction in plan synthesis (Kautz & Selman, 1996; Kambhampati & Yang, 1996; Joslin & Pollack, 1997). The unifying framework and the accompanying insights presented in this paper are expected to be of use to researchers working in all these directions.

The rest of this paper is organized as follows. In Section 2 I review refinement search and show how planning and constraint satisfaction can be modeled in terms of refinement search. In Section 3, I provide a method for doing explanation directed backtracking and explanation based learning in refinement search. In Section 4, I discuss several variations of the basic EDB/EBL techniques produced for the most part by the differing characteristics and requirements of planning and CSP problems, and characterize their tradeoffs. Section 5 show how existing intelligent backtracking and speedup learning algorithms can be seen as the specializations of the EDB/EBL framework. This section also relates failure-driven EBL approaches to pre-processing approaches such as constraint propagation. Section 6 summarizes the contributions of the paper, and speculates on how the improved understanding of EDB/EBL can suggest potentially fruitful avenues of research. Appendix A discusses ways of extending the basic framework to support more flexible backtracking regimes such as dynamic backtracking (Ginsberg, 1993).

2. Refinement Search Preliminaries

The refinement search (also called split-and-prune search (Pearl, 1984)) paradigm is useful for modeling search problems in which it is possible to enumerate all potential solutions (called *candidates*) and verify if one of them is a solution for the problem. Refinement search can be visualized as a process of starting with the set of *all* potential solutions for the problem, and splitting the set repeatedly until a solution can be picked up from one of the sets in *bounded* time. Each search node N in the refinement search thus corresponds to a set of candidates. Syntactically, each search node is represented as a collection of task specific constraints. The *candidate set* of the node is implicitly defined as the set of candidates that satisfy the constraints on the node.

Figure 1 provides a generalized template for refinement search. A refinement search is specified by providing a set of refinement operators (strategies) $\mathbf{R}$, and a solution constructor function `sol`. The search process starts with the initial node $N_\emptyset$, which corresponds to the set of all candidates. The search process involves splitting, and thereby narrowing the set of potential solutions until we are able to pick up a solution for the problem. The splitting process is formalized in terms of refinement operators. A refinement operator $\mathcal{R}$ takes a search node $\mathcal{N}$, and returns a set of search nodes $\{N_1, N_2, \cdots N_n\}$, called refinements of N, such that the candidate set of each of the refinements is a subset of the candidate set of N.

2. Specifically, they use conflict-directed back-jumping, which is equivalent to the explanation-directed backtracking approach we formalize in this paper (see Section 5), and relevance-based learning which provides a syntactic solution to the EBL utility problem (see Section 4)

Algorithm Refine-Node(N)
Parameters: (i) `sol`: Solution constructor function.
(ii) **R**: Refinement operators.

0. **Termination Check:**
If `sol`(N) returns a solution, return it, and terminate.
    If it returns $*fail*$, fail.
    Otherwise, select a flaw F in the node N.
1. **Refinements:**
    Pick a refinement operator $\mathcal{R} \in \mathbf{R}$ that can resolve F.
        (*Not a backtrack point.*).
    Let $\mathcal{R}$ correspond to the n refinement decisions $d_1, d_2, \cdots, d_n$.
    For each refinement decision $d_i \in d_1, d_2 \cdots d_n$ do
        $N' \leftarrow d_i(N)$
        If N' is inconsistent
            Then, fail.
        Else, **Refine-Node**(N').

Figure 1: General template for Refinement search.

Each complete refinement operator can be thought of as corresponding to a set of decisions $d_1, d_2, \cdots, d_n$ such that $d_i(N) = N_i$. Each of these decisions can be seen as an operator which derives a new search node by adding some additional constraints to the current search node.

While refinements split the candidate set of a node, a closely related notion called "*constraint propagation*" narrows the candidate set of the node without splitting it. Sometimes, the presence of certain constraints in the search node, together with the background knowledge (constraints) of task, domain or problem, may imply certain implicit constraints. Constraint propagation essentially derives these constraints and adds them to the node description (thus narrowing its candidate set).

To give a goal-directed flavor to the refinement search, we typically use the notion of "flaws" in a search node and think of individual refinements as resolving the flaws. Specifically, any node N from which we cannot extract a solution directly, is said to have a set of flaws. Flaws can be seen as the *absence* of certain constraints in the node N. The search process involves picking a flaw, and using an appropriate refinement that will "resolve" that flaw by adding the missing constraints. Figure 2 shows how planning and CSP problems can be modeled in terms of refinement search. The next two subsections elaborate this formulation.

2.1 Constraint Satisfaction as Refinement Search

A constraint satisfaction problem (CSP) (Tsang, 1993) is specified by a set of n variables, $x_1, x_2 \cdots x_n$, their respective value domains, $D_1, D_2 \cdots D_n$ and a set of constraints. A constraint $C_i(x_i, \cdots, x_{i_j})$ is a subset of the cartesian production $D_{i_1} \times \cdots \times D_i$, consisting of all tuples of values for a subset $(x_{i_1}, \cdots x_{i_j})$ of the variables which are compatible with each other. A solution is an assignment of values to all the variables such that all the constraints are satisfied. A *binary CSP* problem is one where all the constraints are between exactly

Problem	Nodes	Candidate Set	Refinements	Flaws	Soln. Constructor
CSP, Dynamic CSP	Partial assignment $\mathcal{A}$	Complete assignments consistent with $\mathcal{A}$	Assigning values to variables	Variables needing assignment in $\mathcal{A}$	Checking if all variables needing assignment are assigned and none of the constraints are violated
Planning	Partial plan $\mathcal{P}$	Ground operator sequences consistent with $\mathcal{P}$	Establishment, Conflict resolution	Open conditions, Conflicts in $\mathcal{P}$	Checking if any minimal candidate of $\mathcal{P}$ is a solution

Figure 2: CSP and Planning Problems as instances of Refinement Search

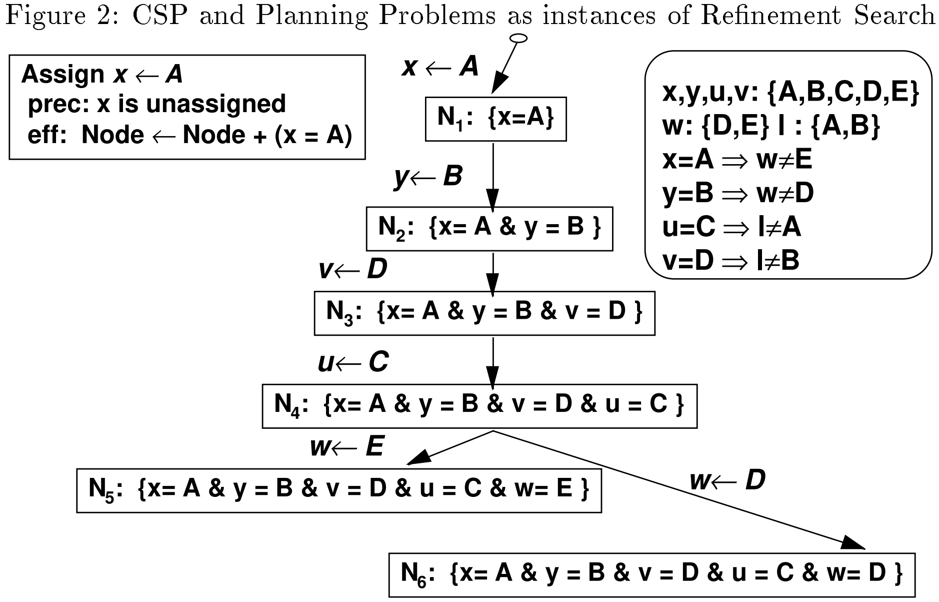

Figure 3: Illustrating CSP as refinement search

two variables. Binary CSPs are interesting because many of the backtracking and learning techniques within CSP are developed with them in mind.

Seen as a refinement search problem each search node in CSP contains constraints of the form $x_i = V_i$, which together provide a partial assignment of values to variables. The candidate set of each such node can be seen as representing all complete assignments consistent with that partial assignment. A solution is a complete assignment that is consistent with all the variable/value constraints of the CSP problem.

Each unassigned variable in the current partial assignment is seen as a "flaw" to be resolved. There is a refinement operator $\mathcal{R}_{x_i}$ corresponding to each variable x_i, which generates refinements of a node N (that does not assign a value to x_i) by assigning a value from D_i to x_i. $\mathcal{R}_{x_i}$ thus corresponds to an "OR" branch in the search space corresponding to decisions $d_1^i, d_2^i, \cdots, d_{|D_i|}^i$. Each decision d_j^i corresponds to *adding* the constraint $x_i = D_i[j]$

to the current partial assignment (where $D_i[j]$ is the j^{th} value in the domain of the variable x_i). We can encode this as an operator with preconditions and effects as follows:

assign($\mathcal{A}, x_i, v_j^{x_i}$)
Preconditions: x_i needs assignment in A.
Effects: $\mathcal{A} \leftarrow \mathcal{A} + (x_i \leftarrow v_j^{x_i})$

Constraint propagation involves deriving consequences of the node assignment constraints, given the background of problem constraints.

Example: Figure 3 illustrates the refinement search process in an example CSP problem. The problem contains five variables, l, x, y, u, v and w. The domains of the variables and the constraints on the variable values are shown in the figure. The search starts by resolving the flaw $needsAssignment(x)$, and then $unassigned(y)$, $unassigned(v)$ and $unassigned(u)$ and $unassigned(w)$ in succession. At the end of the last refinement, two deadend nodes, N_5 and N_6 are produced (a deadend node is one whose partial assignment violates one of the given constraints).

We can use constraint propagation on node N_3 to derive that $l = A$ (since this is a consequence of constraint $v = D$ on the node, coupled with the problem constraint $v = D \Rightarrow l \neq B$, and the constraint that $l = A \vee l = B$. This new constraint can be added to N_3, effectively narrowing its candidate set.

2.1.1 Dynamic CSP as refinement search

There is generalization of CSP problems called Dynamic CSPs (Mittal & Falkenhainer, 1990) that we will find useful in comparing CSP and Planning problems[3]. Just like CSPs, Dynamic CSPs contain variables, their domains, and constraints on legal compound labels. In addition, they also contain a new type of constraints called "activity constraints." Activity constraints are of the following form:

$$x_j = v_j \wedge x_k = v_k \wedge \cdots \wedge x_m = v_m \Rightarrow needsAssignment(x_i)$$

This constraint states that if $x_j, x_k, \cdots x_m$ have the listed values, then the variable x_i will need an assignment.

The initial problem is specified by stating that certain subset of variables require assignments. These are called the active variables. (Contrast this to CSP where all the variables need assignments). The objective is to assign values to all the variables that need assignments, without violating any relevant constraint. Because of the presence of the activity constraints, assigning the original variables may make other currently inactive variables active, adding "$needsAssignment$" flaws corresponding to those variables to the current node. Assignment decisions will thus have the following generic precondition/effect structure:

assign($\mathcal{A}, x_i, v_j^{x_i}$)
Preconditions: x_i needs assignment in A.

3. Some authors (c.f. (Dechter & Dechter, 1988; Verfaillie & Schiex, 1994) seem to use the term Dynamic CSP to refer to CSP problems where the constraints dynamically evolve. In this paper, we shall refer to these later as "incremental" or "evolving" CSPs.

Action	Precond	Add	Dele
Roll(ob)	-	Cylindrical(ob)	Polished(ob) $\wedge$ Cool(ob)
Lathe(ob)	-	Cylindrical(ob)	Polished(ob)
Polish(ob)	Cool(ob)	Polished(ob)	-

Figure 4: Description of a simple job-shop scheduling domain

Effects: $\mathcal{A} \leftarrow \mathcal{A} + (x_i \leftarrow v_j^{x_i})$
If $x_k = v_k \wedge \cdots \wedge x_m = v_m$,
Then, $needsAssignment(x_n)$

In other words, new flaws may result from a refinement decision. Dynamic CSPs were originally proposed to model "configuration" tasks (Mittal & Falkenhainer, 1990). As we shall see below, they can also model planning tasks.

2.2 Planning as Refinement Search

A planning problem is specified by an initial state description I, a goal state description G, and a set of actions A. The actions are described in terms of preconditions and effects, and remain constant for all the problems in a given domain. Both the initial and goal state specifications, and the precondition/effect formulas of actions are described as sentences in some language (usually, functionless first-order predicate logic). The solution is any sequence of actions such that executing those actions from the initial state, in that sequence, will lead us to the goal state.

Figure 4 shows the actions describing a simple jobshop domain. The shop consists of several machines, including a lathe and a roller that are used to reshape objects, and a polisher which is used to polish the surface of a finished object. Given a set of objects to be polished, shaped, etc., the planner's task is to schedule the objects on the machines so as to meet these requirements. A planning problem in this domain might involve polishing an object A and make its surface cylindrical, given that A's temperature is cool in the initial state. Thus, initial state is specified as $Cool(A)$ and the goal state is specified as $Polished(A) \wedge Cylindrical(A)$.

Search nodes in planning can be represented (see (Kambhampati et al., 1995)) as 6-tuples

$$\langle S, \mathcal{O}, \mathcal{B}, \mathcal{L}, \mathcal{E}, \mathcal{C} \rangle,$$

where S is the set of steps, $\mathcal{O}$ is the set of orderings between the steps, $\mathcal{E}$ is the set of effects of the steps in S, $\mathcal{C}$ is the set of preconditions of the steps in S, $\mathcal{B}$ is the set of bindings among the objects taking part in the step effects and preconditions, and finally $\mathcal{L}$ is the set of auxiliary constraints about the truth of conditions over intervals of time. Each step in the plan corresponds to an action in the domain; multiple steps may correspond to the same action. The effects and preconditions of the step are the same as the effects and preconditions of the respective action. The ordering constraints come in two varieties: precedence constraints of the type "$s_1 \prec s_2$", which demand that s_1 precede s_2 in the final solution, but admit any number of actions in between s_1 and s_2; and contiguity constraints of the form "$s_1 * s_2$," which demand that s_1 come immediately before s_2 in the final solution. The special step s_0 is always mapped to the dummy operator `start`, and similarly s_∞ is always mapped to `finish`. The effects of `start` and the preconditions of `finish` correspond, respectively, to the initial state and the desired goals of the planning problem.

Here is an example partial plan for our problem of making A cylindrical and polished in the jobshop domain:

$$\left\langle \begin{array}{c} \mathcal{S}: \{s_0: \texttt{start}, s_1: Roll(A), s_2: Polish(A), s_\infty: \texttt{end}\}, \\ \mathcal{O}: \{(s_0 * s_1), (s_1 \prec s_2), (s_2 \prec s_{infty})\} \\ \mathcal{L}: \{s_0 \overset{Cool(A)}{\rightarrow} s_2, s_1 \overset{Cylindrical(A)}{\rightarrow} s_\infty\} \\ \mathcal{E}: \{effect(s_0, Cool(A)), effect(s_1, Cylindrical(A)), \\ effect(s_2, Polished(A)), effect(s_1, \neg Polished(A)), effect(s_1, Cool(A))\} \\ \mathcal{C}: \{precondition(s_\infty, Cylindrical(A)), precondition(s_\infty, Polished(A)), \\ precondition(s_2, Cool(A))\} \end{array} \right\rangle$$

Notice that initial state and goal state specifications are encoded as the effects and preconditions of s_0 and s_∞ respectively.

The candidates of a partial plan consist of all ground action sequences that are consistent with the constraints of the partial plan. A ground action sequence G is consistent with a partial plan P if G contains all the steps of P (but may also contain other steps), satisfies all the ordering constraints (if $s_1 \prec s_2$ is a constraint in P, and if s_1 is the i^{th} element and s_2 is the j^{th} element in G, then i must be less than j), and all the auxiliary constraints (if $s_1 \xrightarrow{p} s_2$ is a constraint on P, and s_1 and s_2 are the i^{th} and j^{th} elements in G, then none of the actions $G[i+1], G[i+2], \cdots G[j-1]$ must have an effect $\neg p$).[4]

There are several types of complete refinement operators in planning (Kambhampati & Srivastava, 1995), including plan space, state-space, and task reduction refinements. As an example, plan-space refinement proceeds by picking a goal condition and considering different ways of making that condition true in different branches. As in the case of CSP, each refinement operator can again be seen as consisting of a set of decisions, such that each decision produces a single refinement of the parent plan (by adding constraints). As an example, the establishment refinement or plan-space refinement corresponds to picking an unsatisfied goal/subgoal condition p that needs to be true at a step s in a partial plan P (this is referred to as an open condition flaw $precondition(s, c)$ in P), and making a set of children plans $P_1 \cdots P_n$ such that in each plan P_i, there exists a step s' which precedes s and adds the condition p. P_i also contains, (optionally) an auxiliary ("causal link") constraint $s' \xrightarrow{p} s$ to protect p between s' and s. Establishment refinement consists of step addition decisions, which add a new step to establish the condition, and simple establishment decisions which use an existing step to establish the condition. Once again, we can represent these decisions as operators with preconditions and effects. For example, the step addition decision can be written as follows:

StepAddition($effect(s_n, p''), precondition(p', s_d)$)
Use the effect $effect(s_n, p'')$ *of to support the precondition* $precondition(p', s_d)$

Preconditions: $precondition(p', s_d) \in \mathcal{C}$

4. Things are actually slightly more complicated than this since two steps in P may be instances of the same action. To handle this, we need to think in terms of a mapping between steps of P and elements of G, and check the constraints under that mapping (Kambhampati et al., 1995)

$$s' \xrightarrow{p'} s_d \notin \mathcal{L}$$
$$p'' \in \text{effects of } s_n$$

Effects: $\mathcal{S} \leftarrow \mathcal{S} + s_n$

$$\mathcal{L} \leftarrow \mathcal{L} + s_n \xrightarrow{p'} s_d$$
$$\mathcal{O} \leftarrow \mathcal{O} + (s_n \prec s_d) + (0 \prec s_n)$$
$$\mathcal{B} \leftarrow \mathcal{B} + maximal\text{–}general\text{–}unifier(p', p'') + \text{Internal bindings of } s_n$$
$$\mathcal{C} \leftarrow \mathcal{C} + \{precondition(p, s_n) | p \in preconditions\ of\ s_n\}$$
$$\mathcal{E} \leftarrow \mathcal{E} + \{effect(s_n, e) | e \in effects\ of\ s_n\}$$

Its preconditions say that in order to do step addition involving a new step s_n, it must be the case that there is a step s_d which requires a precondition p', s_n has an effect p'' and p' unifies with p''. Once taken, this decision adds the new step s_n to the set of steps in the current plan, makes s_n come after the initial step, and before s_d. Furthermore, bindings are added to make the effect of s_n necessarily unify with the precondition being established. Finally, since s_n is a new step with its own preconditions and effects, the $\mathcal{C}$ and $\mathcal{E}$ fields of the plan are appropriately updated.

A second type of flaw, called an unsafe link flaw, exists whenever the current plan P contains an auxiliary constraint $s_1 \xrightarrow{p} s_2$ and a step s_t such that s_t is not ordered to precede s_1 or follow s_2, and s_t deletes p (s_t is said to threaten the constraint $s_1 \xrightarrow{p} s_2$). The resolution possibilities for this flaw involve promoting s_t to come before s_1 (by adding the ordering relation $s_t \prec s_1$) or demoting s_t to come after s_2 (by adding the ordering relation $s_2 \prec s_t$).

It is interesting to note that while the unsafe link flaws are like "needsAsssignment" flaws in static CSP, the open condition flaws are like "needsAssignment" flaws in dynamic CSP. This is because resolving an open condition may introduce new steps into the plan whose preconditions become new open condition flaws.

Example: We shall now illustrate refinement search process in planning (specifically, partial order causal-link planning, of the type used in SNLP (McAllester & Rosenblitt, 1991)). Figure 5 shows the complete search tree for "polishing and making cylindrical" problem in jobshop domain, discussed earlier. The planner starts with the null plan, and picks up the open condition flaw $Cylindrical(A)@G$. This flaw is resolved by adding the step 1: `Roll`(A) which has an effect $Cylindrical(A)$. The planner then resolves the other open condition flaw $Polished(A)@G$ with the step 2: `Polish`(A). Since the step 1: `Roll`(A), deletes $Polished(A)$, it is now an unsafe-link flaw involving this step and the auxiliary constraint $2 \xrightarrow{Polished(A)} G$. This flaw is resolved by demoting step 1: `Roll`(A) to come before 2: `Polish`(A). The step 2: `Polish`(A) also introduces a new open condition flaw $precondition(Cool(A), 2)$. The planner establishes it using the effects of the initial step 0. Since `Roll`(A) also deletes $Cool(A)$, it threatens this last establishment. When the planner tries to resolve this threat by demoting step 1 to come before step 0, it fails, since 0 already precedes 1. The planner backtracks until the point where it has unexplored alternatives – node A in this example – and explores other possible alternative. It achieves $precondition(Cool(A), G)$ using `Lathe`(A) and then achieves $Polished(A)$ using the operator `Polish`(A). It succeeds in this path and returns a solution.

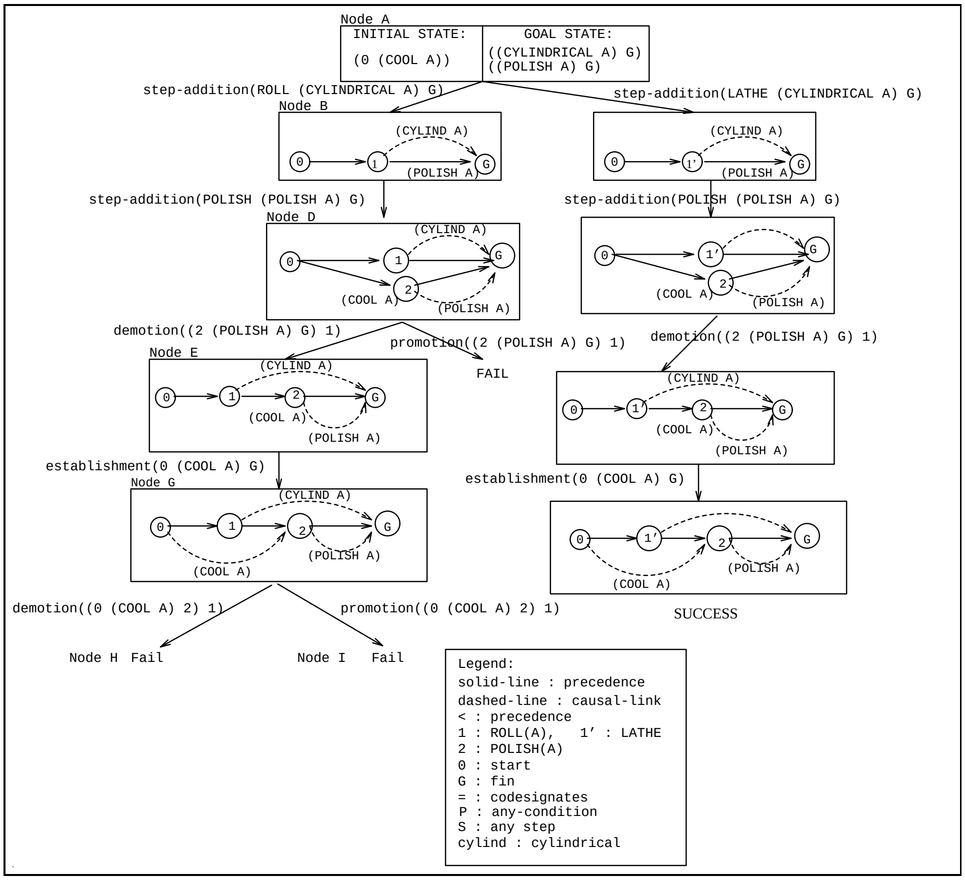

Figure 5: Search Tree illustrating **SNLP** planning process. The figure uses a Lisp-like notation for the plan constraints. Causal link constraints are shown as three element lists, and open conditions and preconditions are shown as two element lists.

2.3 Contrasting planning and CSP

When seen as instances of refinement search, planning and CSP differ in several interesting ways, and these differences shed important light on the applicability of various forms backtracking and learning strategies to these problems.

The most obvious differences are that compared to CSP, in planning the node representations and decisions are more complex. CSP search nodes contain just one type of constraints– assignments to variables. Partial plans contain a variety of constraints, including orderings, bindings and auxiliary constraints. Each refinement decision may lead to several constraints on the partial plan, while in CSP there is an almost one-to-one correspondence between a decision and the assignment constraint that it adds.

CSP problems have a static flaw structure in that refinement search starts with a certain set of flaws ("*needsAssignment*" flaws), and refinement decisions monotonically reduce the number of flaws. This is not the case in planning. We start with a set of open (pre)condition flaws, but in the process of resolving an open condition flaw, we may introduce a new step and all of its preconditions become new open condition flaws. Thus, the number of flaws can both increase and reduce in response to refinement decisions. Furthermore, the set of flaws in a partial plan depends upon the specific set of refinement decisions taken to reach it.

Dynamic CSP problems share the simplicity of node and decision representation with CSPs but have the dynamic flaw structure similar to planning problems (assigning a variable a value may make more variables active, introducing $needsAssignment()$ flaws with respect to them).

In comparing planning and DCSP problems, we notice that while action descriptions implicitly specify how new flaws come into existence in resolving existing flaws, and are thus similar to "activity" constraints, there is no direct analogue in planning for the normal constraints in DCSP. This is because, as formulated, planning problems do not contain any global problem/domain constraints apart from those indirectly imposed by the actions and their precondition/effect description. As we shall see in Section 4.4, this often makes it harder to do an effective failure-based search control of in planning, as the only types of detectable failures involve inconsistencies between the constraints of the plan itself. It also reduces the effectiveness of constraint propagation techniques in planning.[5]

This is however just an artifact of the simplistic nature of the traditional formulation of the classical planning problem. In realistic planning situations, the specification of the domain contains not only the actions, but also a variety of resource and capacity constraints (c.f. (Muscettola, Smith, Cesta, & D'Aloisi, 1991)). Similarly, the problem itself may impose stronger resource and capacity constraints. Thus, a more realistic formulation of planning problem will have global constraints on the plan too.

3. Basic formulation of EDB and EBL

In this section, we will look at the formulation of EDB and EBL in refinement search. The refinement search template provided in Figure 1 implements chronological backtracking by default. There are two independent problems with chronological backtracking. The first problem is that once a failure is encountered, the chronological approach backtracks to the immediate parent and tries its unexplored children – even if it is the case that the actual error was made much higher up in the search tree. The second is that the search process does not learn from its failures, and can thus repeat the same failures in other branches. IB is seen as a solution for the first problem, while EBL is seen as the solution for the second problem. As we shall see below, both of them can be formalized in terms of failure explanations.

We can incorporate IB and EBL within our general refinement search template by (a) computing the explanation of failure at deadend nodes and (b) passing this information over to the "propagate" procedure that effectively computes failure explanations of interior nodes given the explanations at the leaf nodes. The modified refinement search template is shown in Figure 6. The procedure **Propagate** (which works as a co-routine to refinement

5. There does exist a rather high-level constraint propagation in planning. In (Kambhampati & Yang, 1996), we argue for separating the notions of splitting and narrowing of candidate sets in the context of planning, and show that traditional plan refinement strategies narrow the candidate set of the plan, even if we choose not to split the result of the refinement into multiple partial plans. Thus, these refinements can be seen to be doing a sort of constraint propagation. Specifically, they can be seen as explicating the consequences of partial plan constraints, given the task level constraints of planning, and the domain level theory of actions.

Algorithm Refine-Node(N)
Parameters: (i) `sol`: Solution constructor function. (ii) $\mathbf{R}$: Refinement operators.

0. **Termination Check:**
If `sol`(N) returns a solution, return it, and terminate.
  If it returns $*fail*$, fail.
  Otherwise, select a flaw F in the node N.
1. **Refinements:**
  Pick a refinement operator $\mathcal{R} \in \mathbf{R}$ that can resolve F.
    (*Not a backtrack point.*).
  Let $\mathcal{R}$ correspond to the n refinement decisions $d_1, d_2, \cdots, d_n$.
  For each refinement decision $d_i \in d_1, d_2 \cdots d_n$ do
    $N' \leftarrow d_i(N)$
    If N' is inconsistent
      Then, fail.
      Set $E(N') \leftarrow$*an explanation of failure of* N'
      Call **Propagate**(N')
    Else, **Refine-Node**(N').

Figure 6: Refinement search augmented with IB and EBL capabilities. The main augmentation involves computing the explanation of failure of the deadend node and, and analyzing it (using "propagate" routine).

search) is shown in Figure 7. An approximate flow chart of the procedure is shown in Figure 8. In the following we explain how the IB and EBL capabilities follow from the propagate procedure.

Suppose a search node N is found to be *failing* by the refinement search template in Figure 1. To avoid pursuing refinements that are doomed to fail, we would like to backtrack *not* to the immediate parent of the failing node, but rather to an ancestor node N' of N such that the decision taken under N' has had some consequence on the detected failure. To implement this approach, we need to sort out the relation between the failure at N and the refinement decisions leading to it. We can do this by declaratively characterizing the failure at N.

3.1 Explaining Failures

From the refinement search point of view, a search node N is said to be **failing** if its candidate set provably does not contain any solution. This can happen in two ways– the more obvious way is when the candidate set of N is *empty* (because of an inconsistency among the constraints of N), or because the constraints of N together with the global constraints of the problem or domain, and the requirements of the solution, are inconsistent. For example, in CSP, a partial assignment $\mathcal{A}$ may be failing because $\mathcal{A}$ assigns two values to the same variable, or because the values that $\mathcal{A}$ assigns to its variables are inconsistent with the some of the specific constraints. Similarly, in the case of planning, a partial plan P may be inconsistent because of inconsistent causal commitments (a causal link $s' \xrightarrow{C} s$ is inconsistent if the plan contains an action s'' that deletes C and $s' \prec s'' \prec s$), ordering

Procedure Propagate(N_i)

$parent(N_i)$: The node that was refined to get N_i.
$d(N_i)$: decision leading to N_i from its parent;
$E(N_i)$: explanation of failure at N_i.
$F(N_i)$: The flaw that was resolved at this node.

1. $E' \leftarrow$ `Regress`$(E(N_i), d(N_i))$
2. If $E' = E(N_i)$, then (dependency directed backtracking)
   $E(parent(N_i)) \leftarrow E'$; **Propagate**$(parent(N_i))$
3. If $E' \neq E(N_i)$, then
   3.1. If there are unexplored siblings of N_i
      3.1.1 Make a rejection rule R rejecting the decision $d(N_i)$, with E' as the rule antecedent. Store R in rule set.
      3.1.2. $E(parent(N_i)) \leftarrow E(parent(N_i)) \wedge E'$
      3.1.3. Let N_{i+1} be the first unexplored sibling of node N_i.
      Refine-node(N_{i+1})
   3.2. If there are no unexplored siblings of N_i,
      3.2.1. Set $E(parent(N_i))$ to
      $E(parent(N_i)) \wedge E' \wedge F(parent(N_i))$
      3.2.3. **Propagate**$(parent(N_i))$

Figure 7: The complete procedure for propagating failure explanations and doing dependency directed backtracking

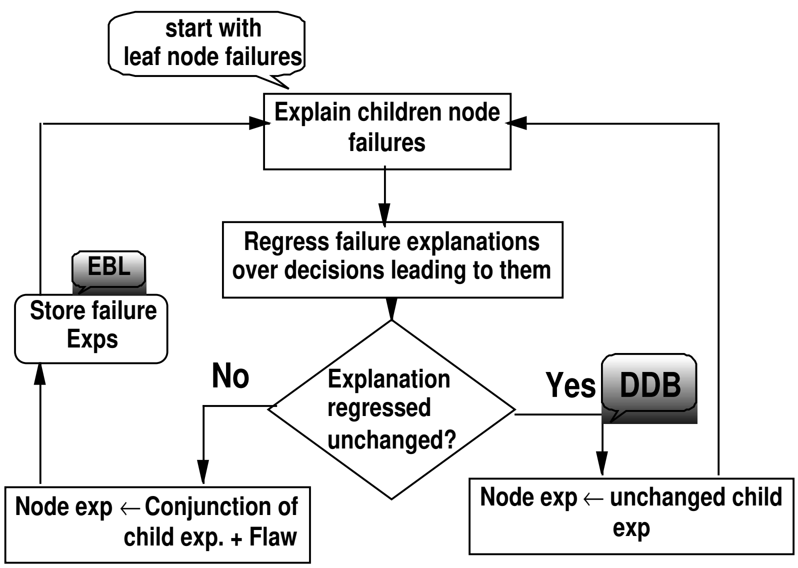

Figure 8: An approximate flow chart of propagation procedure. See Figure 7 for the complete procedure.

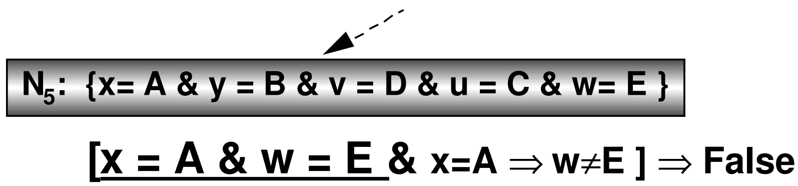

Figure 9: Computing explanation of failure of a leaf node

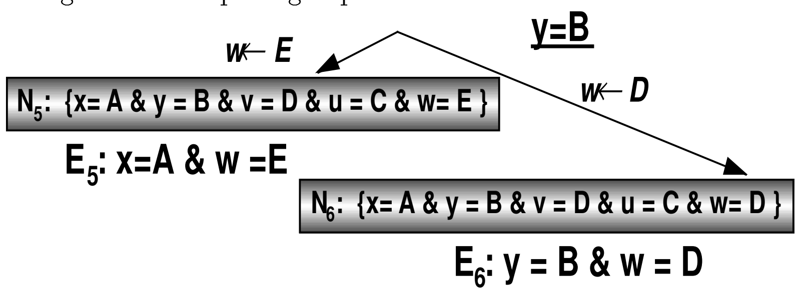

Figure 10: Regressing a failure explanation over a decision

cycles or binding inconsistencies, or because the plan violates a domain constraint (such as resource or capacity constraints).

In either case, we can associate the failure at N with the constraints in N, say E, which possibly together with the domain or problem constraints, Δ, lead to an inconsistency (i.e., $\Delta \wedge E \models False$). The problem and domain-independent characterization of this failure would be $\Delta \wedge E$. However, we normally consider E to be the explanation of failure of N since presumably the constraints in Δ are part of the problem (see Section 4.1 for a discussion of when the more conservative failure explanation will be needed).

For the CSP example problem shown in Figure 3, the search fails first at node N_5. Figure 9 shows the full explanation of failure of node N_5. It is $x = A \wedge w = E$ (since this winds up violating the first constraint). The more conservative failure explanation will be $x = A \wedge w = E \wedge (x = A \Rightarrow w \neq E)$.

3.2 Regression

Given a failing node, in order to support intelligent backtracking, we need to evaluate the effect of various decisions leading to that node on the failure. Further, having picked an ancestor node to backtrack to, we will be interested in finding which constraints of the node are indirectly responsible for the observed failure (since this information can help in computing the explanation of failure of this ancestor node if the rest of the branches under the node were also to fail). Since failures are characterized in terms of the failure explanations, this process involves checking which constraints in the failure explanation are added by which decisions leading to the failing node. Formally, this is accomplished by regressing failure explanations over decisions.

Formally, regression of a constraint c over a decision $d(N)$ is the set of constraints that must be present in the partial plan before the decision $d(N)$, such that c is present after taking the decision.[6] Regression of this type is typically studied in planning in conjunction

6. Note that in regressing a constraint c over a decision $d(N)$, we are interested in the weakest constraints that need to be true before the decision so that c will be true after the decision is taken. The preconditions

with backward application of STRIPS-type operators (with add, delete, and precondition lists), and is quite well-understood (see (Nilsson, 1980)).

There are two changes that we need to impose to make it useful in our situation. First, we need to adapt regression to refinement decisions (as against actions). This should be straightforward given our characterization of refinement decisions in terms of preconditions and effects. Second, we need to specialize the regression process to give us the conditions in the parent node of N that lead to the presence of constraint c after the decision d (rather than simply the weakest constraints that will ensure c after d). We shall call this specialized version "example guided regression."

Specifically, example guided regression of a constraint c over a decision d taken from a node N can be formally defined as follows:

$$
\begin{aligned}
&\mathbf{Regress}(c, d, N) && \\
&\quad = \text{True} && \text{if } c \in effects(d) \\
&\quad = \text{False} && \text{if } \neg c \in effects(d) \\
&\quad = C' : \{c_1 \cdots c_j\} && \text{if } C' \subseteq N \text{ and } (effects(d) \wedge C') \vdash c \text{ and} \\
& && \quad N \not\vdash c \text{ and } \forall_{C''} C'' \wedge effects(d) \vdash c \text{ iff } C' \subseteq C'' \\
&\quad = c && \text{Otherwise}
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{Regress}(c_1 \wedge c_2 \cdots \wedge c_n, d, N) \\
&\quad = Regress(c_1, d) \wedge Regress(c_2, d) \cdots \wedge Regress(c_n, d)
\end{aligned}
$$

In words, regressing a constraint c over a decision d results in $True$ if the constraint c is added by d, $False$ if it is deleted by d. It results in the set of constraints C' if c can be inferred from the effects of d and the set of constraints C' that are present in N, c couldn't have been inferred from the constraints of N alone (without $effects(d)$), and no proper subset of C' in combination with $effects(d)$ can allow us to infer c. Notice that there may be multiple minimal constraint sets that satisfy the restrictions on C'. In such a case, we consider any one of them the result of regression. In all other cases, c regresses to itself. The regression of a failure explanation is accomplished by regressing individual constraints comprising the failure explanation and conjoining the results.

The third clause in the definition of regression corresponds to the case where c is an inferred constraint. That is, c is inferred from the effects of d and some minimal set of constraints C' already present in the node N on which the decision is taken. It is the insistence on C' being a subset of N that makes the regression "example guided." In contrast, normal regression can consider any minimal set C'' that allows us to infer c. Inferred constraints come into refinement search when refinements are interspersed with constraint propagation (such as forward checking or local consistency enforcement (Tsang, 1993)).

Figure 10 illustrates regression of the failure explanation of N_5, $x = A \wedge w = E$ over the decision "$w \leftarrow E$" that leads to N_5, resulting in $x = A$ (since $w = E$ is the only constraint that is added by the decision).

To illustrate regression of inferred constraints, suppose we had three additional constraints in this example problem:

$$v = D \wedge u = C \wedge w = E \Rightarrow l = A$$

of the decisions must hold in order for the decision to have been taken any way, and thus do not play a part in regression.

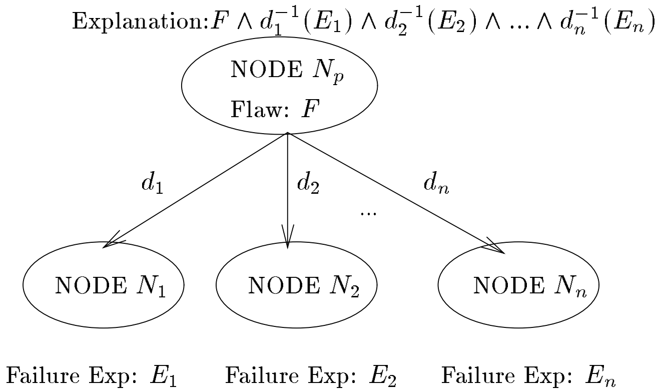

Figure 11: Computing Failure Explanations of Interior Nodes

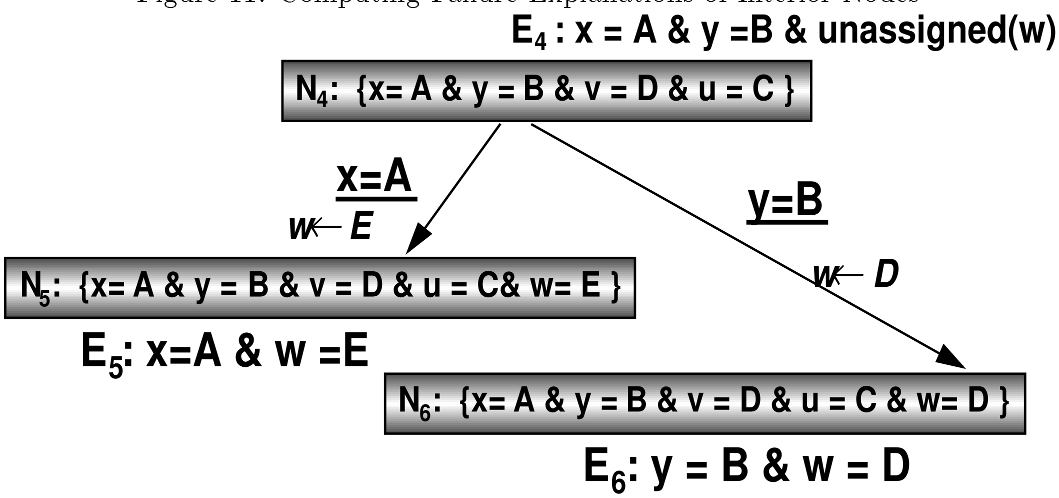

Figure 12: Example of interior node explanation computation

$$v = D \wedge x = A \wedge y = B \wedge w = E \Rightarrow l = A$$
$$v = A \wedge w = E \Rightarrow l = A$$

The result of doing example guided regression of the constraint $l = A$ over the decision $w \leftarrow E$ taken on N_4, will be either $v = D \wedge u = C$ or $v = D \wedge x = A \wedge y = B$. (Although $v = A \wedge w = E$ also derives $l = A$, it cannot be the result of example guided regression since it is not true in the current assignment at N_4. In contrast, the weakest condition for ensuring $l = A$, as given by the normal regression, will be the disjunction of constraint sets

$$[l = A] \vee [v = D \wedge u = C] \vee [v = D \wedge x = A \wedge y = B] \vee [v = A \wedge w = E]$$

3.3 Computing Explanations of failures of Interior Nodes

If the explanation of failure changes after regression, i.e., $E' = d^{-1}(E) \neq E$, then we know that the decision leading to N did have an effect on the failure in N. At this point, we need to consider the sibling decisions of d under N_p. If there are no unexplored sibling decisions, this again means that all the refinements of N_p have failed. The failure explanation for N_p

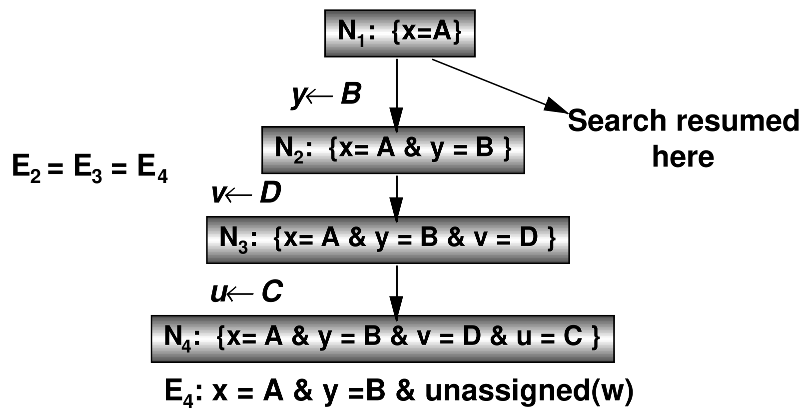

Figure 13: Illustration of dependency directed backtracking

can be computed in terms of the failure explanations of its children, and the flaw that was resolved from node N_p, as shown in Figure 11.

Intuitively, this says that as long as the flaw exists in the node, we will consider the refinement operator again to resolve the flaw, and will fail in all branches.[7] The failure explanation thus computed can be used to continue propagation and backtracking further up the search tree.

In the Propagate procedure shown in Figure 7, the interior node explanations are computed incrementally by accumulating the regressed failure explanations from each of the branches under it. Once the last branch has been explored, the accumulated regressions are conjoined with the flaw description and propagated upwards. (An exception occurs if the failure explanations of any of the branches regress unchanged; causing EDB. See Section 3.4)

Figure 12 illustrates how the explanation of failure of interior nodes is computed in our running example. N_6 is also a failing node, and its explanation of failure is $y = B \wedge w = D$. When this explanation is regressed over the corresponding decision, we get $y = B$. This is then conjoined with the regressed explanation from N_5, and the flaw description at N_4 to give the explanation of failure of N_4 as $E(N_4) : x = A \wedge y = B \wedge needsAssignment(w)$. The more conservative explanation of failure will also name the constraints whose violation caused the failure, and will be:

$$x = A \wedge y = B \wedge needsAssignment(w) \wedge (x = A \Rightarrow w \neq E) \wedge \quad (1)$$
$$(y = B \Rightarrow w \neq D) \wedge (w = E \vee w = D) \quad (2)$$

3.4 Explanation Directed Backtracking

Returning to our earlier discussion, suppose the result of regressing the explanation of failure E of node N over the decision d leading to N, $d^{-1}(E)$, be E'. Suppose $E' = E$. In such a case, we know that the decision d did not play any role in causing this failure.[8] Thus,

7. It is interesting to note that we conjoin the failure explanations of the children branches with the description of the flaw that is being resolved by all those branches, rather than the preconditions of the decisions that attempt to resolve it. This makes sense because as long as the flaw is present, these decisions will be used to resolve it, thus resulting in a failure.

8. Equivalently, IB can also be done by backtracking to the highest ancestor node of N which still contains all the constraints in E. I use the regression based model since it foregrounds the similarity between IB and EBL.

there is no point in backtracking and trying another alternative at N_p. This is because our reasoning shows that the constraints comprising the failure explanation E are present in N_p also, and since by definition E is a set of inconsistent constraints, N_p is also a failing node. This reasoning forms the basis of *explanation directed backtracking*. Specifically, in such cases, we can consider N_p as failing, setting E as its explanation of failure (in the process trashing any accumulated information about the failure explanation of N_p),and continue backtracking. This is what the **propagate** procedure in Figure 7 does.

Figure 13 illustrates explanation directed backtracking in our running example. Having computed the explanation of failure of node N_4, we continue the propagation process upwards. Now, the decision $v \leftarrow D$ does not affect the failure explanation N_4, and thus we backtrack over node N_3, without refining it further. Similarly, we also backtrack over N_2. $E(N_4)$ does change when regressed over $y \leftarrow B$ and thus we restart search under N_1.

To support EDB, we just need to keep track of the partial failure explanations at each of the ancestor nodes on the current search branch. If d is the maximum depth of the search tree, we will store $O(d)$ partial explanations. The size of each partial explanation is at most the size of the search node (since explanations must be subset of nodes). If the size of the largest node (partial plan, partial assignment etc.) is S_n, the total space used by EDB is $O(dS_n)$. For the case of CSP, d and S_n are both $O(n)$, where n is the number of variables, leading to $O(n^2)$ space overhead.

3.5 Explanation Based Learning

Until now, we talked about the idea of using failure explanations to assist in intelligent backtracking. The same mechanism can however also be used to facilitate what has traditionally been called EBL. Specifically, suppose we found out that an interior node N_p is failing (possibly because all its children are failing), and we have computed its explanation of failure E_p. Suppose we remember E_p as a "learned failure explanation." Later on, if we find a search node N' in another search branch such that E_p is a subset of N', then we can consider N' to be failing with E_p as its failure explanation. A variation of this approach involves learning search control rules (Minton et al., 1989) which recommend rejection of individual decisions of a refinement operator if they lead to a failing node. When the child N_1 of the search node N_p failed with failure explanation E_1, and $E' = d^{-1}(E_1)$, we can learn a rule which recommends rejection of the decision d whenever E' is present in the current node. In other words, search control rules are nothing but failure explanations re-expressed in a syntactically different way.

In our CSP example, after computing the explanation of failure of N_4 to be $x = A \wedge y = B \wedge needsAssignment(w)$, we can remember this as a learned failure explanation (aka nogood (Stallman & Sussman, 1977)), and use it to prune nodes in other parts of the search tree.

Unlike EDB, whose overheads are generally negligible compared to chronological backtracking, learning failure explanations through EBL has two types of hidden costs. First, there is the storage cost. If we were to remember every learned failure explanation, the storage requirements can be exponential because each leaf node in the search tree may potentially be failing. Next, there is the cost of *using* the learned failure explanations. Since in general, using failure explanations will involve matching the failure explanations (or the antecedents of the search control rules) to the current node, the match cost increases as the number of stored explanations increase. This problem has been termed the "EBL Utility

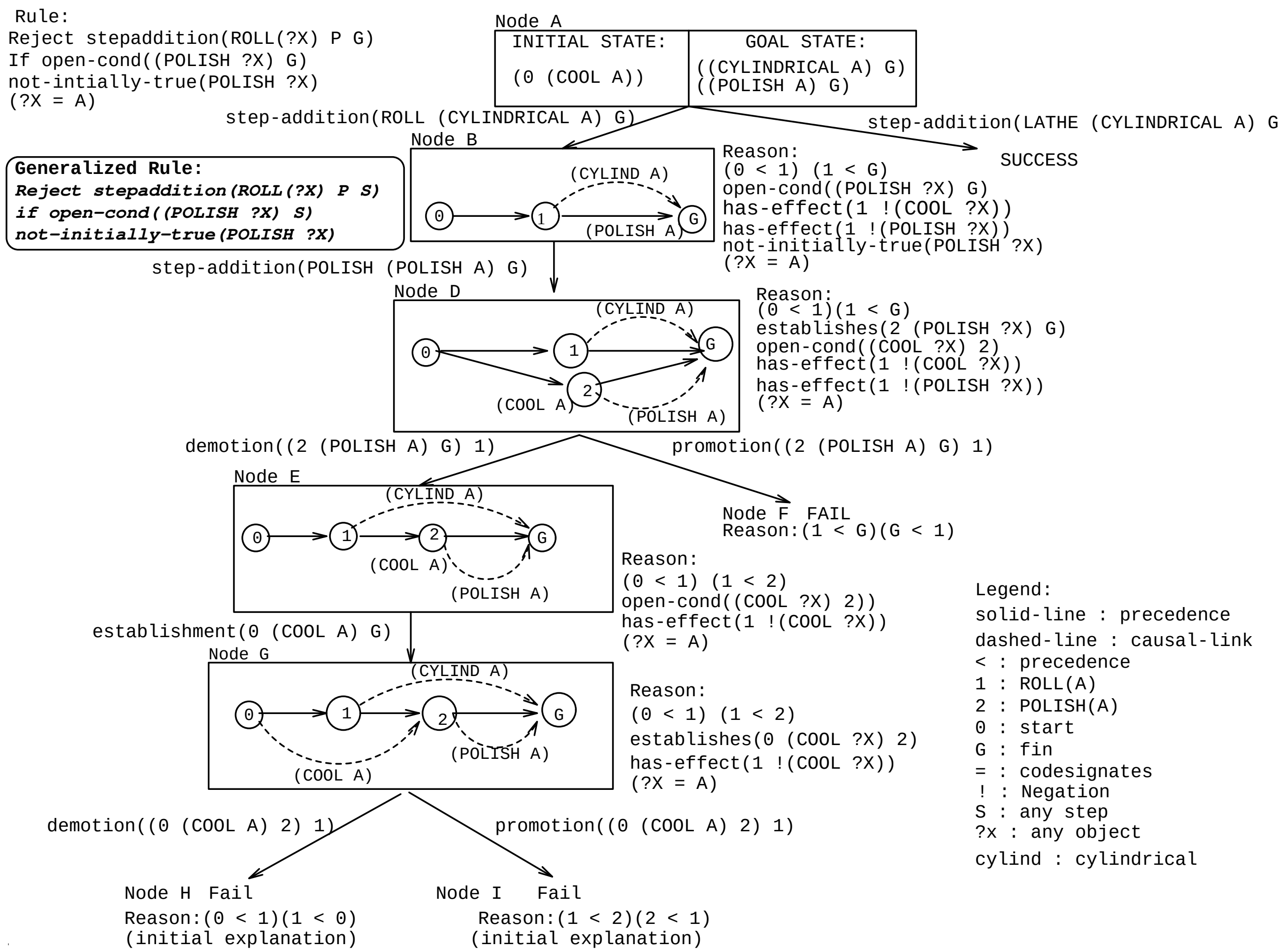

Figure 14: A complete analysis of failures in the Job-shop Scheduling example

Problem" in the Machine learning community[9] (Minton., 1990; Gratch & DeJong, 1992). We shall review various approaches to it later.

3.6 Examples illustrating EDB and EBL in Planning

In this section, we illustrate the EDB and EBL ideas with a series of planning examples. These will complement the CSP examples we used until now, and will reinforce the generality of our treatment of EBL and EDB. Let us consider again the example planning search tree shown in Figure 5. Figure 14 shows the complete trace of the propagation of explanations in this example.[10] When the planner failed at node H and I in the Figure 14, the failures are explained in terms of ordering inconsistencies as shown in the figure. Specifically, the explanation of failure at node H is $(0 \prec 1) \wedge (1 \prec 0)$, and that at node I is $(1 \prec 1) \wedge (2 \prec 1)$. When we regress the explanation of node H over the demotion decision

9. In some ways, this is misleading as it seems to suggest that only explanation based learning can suffer from utility problem. Any approach for learning control information, whether it is explanation based or inductive (c.f. (Estlin & Mooney, 1996), could suffer from the utility problem.

10. See (Kambhampati et al., 1996) for a more comprehensive treatment of EBL/EDB in partial order planning.

that was used to resolve the unsafe causal link flaw involving $0 \overset{Cool(A)}{\rightarrow} 2$ and the effects of step 1 ($demotion(effect(1, \neg Cool(A)), 0 \overset{Cool(A)}{\rightarrow} 2)$), it results in the ordering constraint $(0 \prec 1)$ (since the demotion only adds ordering constraints). Similarly when we regress the explanation of node I over the $promotion(effect(1, \neg Cool(A)), 0 \overset{Cool(A))}{\rightarrow} 2)$, it results in the ordering constraint $(1 \prec 2)$. Now, at node G, we have the explanations for the failure of the branches H and I. Thus, the explanation at node G (also shown in Figure 14) is:

$$\begin{aligned} E(G) &= Constraints\ describing\ the\ Unsafe\ link\ flaw\ \wedge (0 \prec 1) \wedge (1 \prec 2) \\ &= (0 \overset{Cool(A)}{\rightarrow} 2) \wedge effect(1, \neg Cool(A)) \wedge (1 \not\prec 0) \wedge (2 \not\prec 1) \wedge (0 \prec 1) \wedge (1 \prec 2) \\ &= (0 \overset{Cool(A)}{\rightarrow} 2) \wedge effect(1, \neg Cool(A)) \wedge (0 \prec 1) \wedge (1 \prec 2) \end{aligned}$$

The last step follows from the simplification $(s_1 \not\prec s_2) \wedge (s_2 \prec s_1) \equiv (s_2 \prec s_1)$ (since $(s_2 \prec s_1)$ implies $(s_1 \not\prec s_2)$ in any consistent plan). This explanation can be interpreted as follows: if there are three steps s_0, s_1 and s_2 such that $(s_0 \prec s_1) \wedge (s_1 \prec s_2)$ and if a causal link $s_0 \overset{Cool(A)}{\rightarrow} s_1$ is threatened by the step s_1, prune the node from search space. This type of propagation is continued all the way up the tree, learning the failure explanation of node B as shown to the right of B in Figure 14. This can be converted into a "pruning rule" of the following form:

> **If** $(s_0 \prec s_1) \wedge (s_1 \prec G)\ \wedge$
> $precondition(Polished(A), G) \wedge$
> $effect(s_1, \neg Cool(A)) \wedge$
> $effect(s_1, \neg Polished(A)) \wedge$
> $\neg$`initially–true`$(Polished(A))$
> **then Reject** the plan

Since we reached node B by using a step addition decision, we can also regress this explanation over the step addition decision leading to B, and write the result as the premise of a search control rule prohibiting the step addition decision:

> **If** $(s_0 \prec s_1) \wedge (s_1 \prec G)\ \wedge$
> $precondition(Polished(A), G) \wedge$
> $\neg$`initially–true`$(Polished(A))$
> **then Reject** $stepaddition(Roll(A), precondition(cylindrical(A), G))$

There is no instance of EDB skipping over intermediate decisions in the example shown in Figure 14. To illustrate it, we give another simple planning example, shown in Figure 15. Here, the first plan contains two open condition flaws $P1@G$ and $P2@G$ respectively. The first refinement involves resolving the flaw $P1@G$, and this is done in the left branch by the step addition decision that adds the step $O1$ which requires $Q1$ and $Q2$ conditions. Next, the flaw $Q1@O1$ is resolved to give rise to the plan at node D. At this point, the flaw $Q2@O1$ is chosen and the planner finds that there are no steps giving $Q2$, and $Q2$ is not initially true. The explanation of failure of node D is thus computed as $Q2@O1 \wedge \neg$`initially–true`$(Q2)$.[11] Since this explanation of failure does not change after regression over the step addition

11. See Section 4.1 for more discussion on the presence of `initially–true`() constraints.

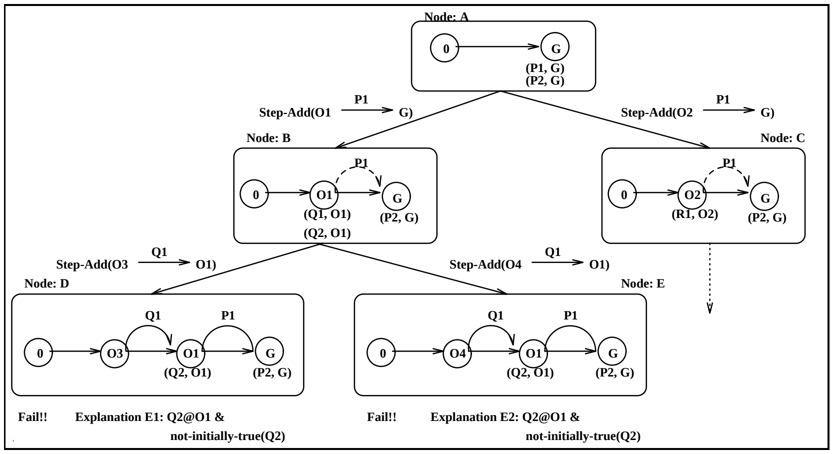

Figure 15: An example for dependency directed backtracking

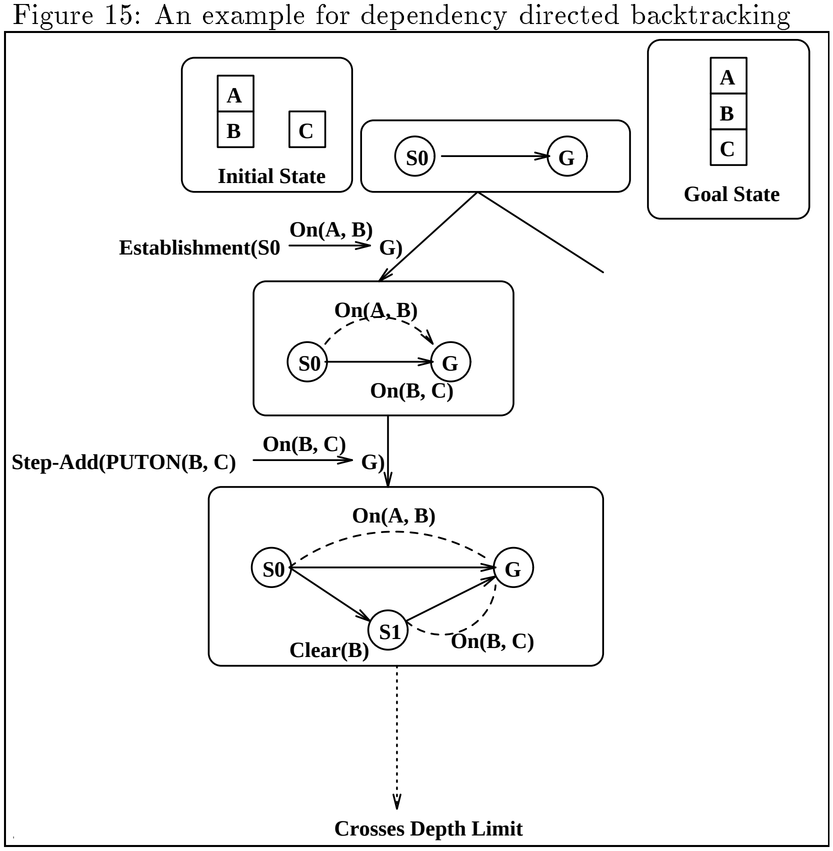

Figure 16: An example showing a branch of a search tree whose failure can only be explained in terms of violation of domain constraints.

decision, the planner can prune the other sibling of the node D, i.e. node E, and continue the propagation of explanation above node B with the failure explanation of B set to the same as that of D.

Finally, all the failures we saw in these examples above are with respect to search node constraints. Some times the failure may be explainable only with respect to domain constraints. To illustrate this type of failure, consider the example in Figure 16, where the bottom-most plan in the search tree has no ordering or binding inconsistencies. However, we can use the domain constraint of the blocks world, that a block can't both be clear and have another block on top of it, to detect an implicit failure in this plan. Specifically, in this plan, in the state preceding the step $S1$, we must have both $Clear(B)$, which is a precondition of $S1$, and the condition $On(A, B)$ which is being protected over the length of the plan, true. Clearly, this is impossible. Thus, we can have as failure explanation for this node:

$$0 \overset{On(A,B)}{\rightarrow} G \wedge (0 \prec 1) \wedge (1 \prec G) \wedge precondition(Clear(B), 1) \wedge (B \not\approx Table)$$

4. Tradeoffs and Variations on the basic EDB and EBL Theme

The basic approach to EDB and EBL that we described in the previous section admits many variations based on how the explanations are represented, selected and remembered. I discuss these variations below.

4.1 Contextualizing Explanations of Failure

Logically, the explanation of failure is a formula ϕ such that $\phi \models False$. However, often, it is useful to separate ϕ into search node specific and search node independent (but problem or domain dependent) parts, and consider only the former as the explanation of failure. For example, suppose Δ is the background knowledge of the problem, task or domain that is independent of the particular node, where the failure occurred. We may then remove from ϕ those parts that are present in Δ, thereby getting ϕ'. We now have $\phi' \wedge \Delta \models False$. In as much as Δ remains constant between the current node and the context where we want to use the explanation, we are safe in considering only ϕ' as the explanation of failure.

Let me give examples to illustrate problem, task and domain knowledge. In planning, the problem knowledge may involve the actions, as well as any problem-specific resource constraints (e.g., a particular goal needs to be achieved before time t_1). The task knowledge is knowledge about planning tasks in general, and will include axioms such as "no step can precede as well as follow another step", "no variable can have two values" etc, as well as theories of looping (Kambhampati, 1995). The domain knowledge is domain level resource and capacity constraints, such as "no block can have more than one block on top of it". In the case of CSP, problem and domain knowledge consists of constraints on compound labels. The domain knowledge may refer to the domain level constraints, and will not change from problem to problem. Task knowledge includes axioms such as "no variable can have more than one assigned value."

Broadly speaking, if we know the range of situations where we hope to use the set of failure explanations we learn from the current problem, we can then use that knowledge to "contextualize" and shorten the explanations, by stripping off the aspects of the explanations that are guaranteed to hold constant over the range of those situations. This is the reason we use $(x = A \wedge w = E)$ as the explanation of failure for the node N_5 in Figure 9, and do not explicitly mention the constraint $(x = A \Rightarrow w \neq E)$ that is violated (since the nogood is used only in other branches of the same search tree, and the constraint will be active in those branches too).

Such a contextualization of explanations can also be done for interior nodes. For example, Figure 11 shows that in general the explanation of failure of an interior node N_p is computed by conjoining the regressed explanations of failures of the children nodes with the description of the flaw that was resolved at N_p. However, in CSP problems, learned explanations are often used only in other branches of the same problem. Since in standard CSP, the search attempts to assign the same variables in all the branches, the flaw structure remains constant, and thus we can remove it from the description of the interior node explanation. This is what is done in most CSP learning approaches, whereby the flaw description $needsAssignment(w)$ is will not be attached to the explanation of failure of the node N_4.

This argument does not hold if we are dealing with any of the following variations of standard CSP problems:

1. Dynamic constraint satisfaction problems (Mittal & Falkenhainer, 1990) (see Section 2.1.1) where flaw structure evolves as refinements take place.

2. Incremental or evolving CSPs, where the constraints are dynamically added or removed as the problem evolves (Dechter & Dechter, 1988; Schiex & Verfaillie, 1993; Verfaillie & Schiex, 1994)

3. Realistic CSPs where a subset of problem constraints are constant from one problem to another (see Section 6)

In the first case, the interior node failure explanations must contain the flaw description in order that they can be used in other search branches of the same problem, as the flaw structure evolves with refinement decisions and can be different in different branches (see Section 2.1.1). In the second case, the failure explanations must contain the specific problem constraints that are being violated, since they may become invalid later when the constraints are relaxed.[12]

In the third case, inter-problem transfer is possible. However, both the flaw description and the violated constraints must be part of the failure explanations since the new problem may require assignment for a different (but overlapping) set of variables, and may involve different (but overlapping) set of constraints. To illustrate this, consider the explanation of failure of the interior node N_4 in Figure 12. The complete explanation of failure is:

$$x = A \wedge y = B \wedge needsAssignment(w) \wedge (x = A \Rightarrow w \neq E) \wedge \\ (y = B \Rightarrow w \neq D) \wedge (w = E \vee w = D)$$

Since w must be assigned in any solution for this problem, for intra-problem learning, the flaw description is typically stripped from the explanation. However, if we were to use this explanation in a different problem, the fact that $x = A \wedge y = B$ leads to inconsistency only if z needs to be assigned, and only when $(x = A \Rightarrow w \neq E)$ and $(y = B \Rightarrow w \neq D)$ are valid constraints in the problem, is very crucial to make the failure explanation sound.

12. Schiex and Verfaillie (Schiex & Verfaillie, 1993; Verfaillie & Schiex, 1994) call explanations of failure that name violated constraints "justified explanations of failure", and argue that these are required for supporting backtracking and learning in CSPs where constraints evolve dynamically. Our discussion is more general as it points out that sometimes even the flaw description needs to be added to the failure explanation.

In the case of planning, the nogoods/rules learned in one problem are commonly expected to be used in other problem situations. This means that any aspect of the constraints resulting in failure that may change from problem to problem must be kept as part of the node explanation. For example, when the node D fails in Figure 13 because the flaw $Q2$ cannot be resolved, the complete reason for failure can be written as the conjunction of three clauses:

1. There is a flaw $Q2@O1$ in the plan.
2. The condition $Q2$ is not present in the initial state of the problem.
3. There are no operators in the domain which can give $Q2$.

Which of these clauses should be part of the failure explanation depends upon whether we expect to use the failure explanations (and rules learned) in (a) this branch alone (b) other branches of this problem and (c) other problems of this domain or (d) other domains of this type. Clearly, as we go from a to d , we are trying to increase the coverage of our analysis, and thus the explanation needs to be qualified more carefully.

Of the three clauses causing the failure, the first clause must be part of the failure explanation if we want to use the learned nogoods in scenarios b,c,d. This is because, the flaw $Q2@O1$ is specific to this particular search branch and may not occur in another branch (where presumably $P1$ may be established by the use of an operator that doesn't need $Q2$) or another problem (where the goal $P1$ itself may not be present). In the latter two cases, the fact that $Q2$ is unestablishable may not lead to a failure.

The second clause can be skipped as long as the intended usage of learned rules is in scenarios (a) and (b) (intra-problem learning). But, if we allow the use of the rule in different problems, then since the initial state changes from problem to problem, we must add a clause to the effect that initial state does not give the condition $Q2$ (alternately, we must do counter-factual search to see if the failure would have occurred even if the initial state gave the condition $Q2$ (Kambhampati et al., 1996)). This is illustrated by the use of ¬`initially−true`() clauses in the failure explanations in the examples in Figures 14 and Figure 13.

Finally, the third clause, that none of the operators give $Q2$, can be skipped unless the intended usage is the scenario d, since operators are assumed to remain the same as long as we stick to the same domain. Normally, in planning and problem solving, the learned rules are expected to be used in other problems in the same domain. Thus, normally, we make the clauses 1 and 2 part of the failure explanation, while skipping the third clause.

4.2 Explanation Generalization

An issue closely related to explanation contextualization is the "explanation generalization". When we expect to use a failure explanation in problems other than the one in which it was produced, it is often worth trying to see if the failure explanation is specific to the "objects" (steps, constants) involved in the current problem, or whether it remains the same even if the objects change. In the job-shop scheduling example shown in Figure 14, the normal EBL/EDB analysis at node A tells us that we can reject the $ROLL$ operator (see the rule to the right of node A in the figure) to make the part A cylindrical as long as we have to keep A polished at the end ($Polish(A)@G$). However, it is clear that the failure has nothing to do with the exact identity of the part A. Even if we are trying to make another part C cylindrical and polished, we still should be able to use this rule to rejct $ROLL$ operation. What is more, the failure also has nothing to do with the specific identity of the step G. It would have occured even if the $Polished(A)$ and $Cylindrical(A)$ were preconditions of some intermediate step s_1 (instead of being toplevel goals). In other words, we can substitute variables for step names and object names in the failure explanations. The boxed rule in Figure 14 shows such a generalized rule.

In general, variablizing every constant may not lead to sound explanations of failure. The correctness of the generalization needs to be checked by a verifying that the proof of failure of a particular node will still be valid after generalization. In practice, this verification proof can be avoided by pre-processing the domain knowledge. For example, it has been shown (Kambhampati et al., 1996; DeJong, 1996) that variablizing all objects in the failure explanation (taking care to have the same variable substitutes a specific constant everywhere in the explanation) is a sound strategy when the domain theory (actions, domain constraints etc.) is "*name-insensitive*", in that it does not refer to specific objects by name. For example, a precondition $On(x, Table)$ is name-sensitive while the preconditions $On(x, t) \wedge Table(t)$ is name-insensitive. Using similar arguments, in the case of steps, the only step that is referred to directly by name by the planning decisions (such as promotion, demotion, step-addition) is the initial step. For example, step-addition decision adds an ordering constraint between the newly introduced step and the initial step. So, all step-names other than initial step name can be generalized without worrying about losing soundness. (Even initial step can be generalized if an ordering involving it is never regressed over a step-addition decision that added it, see (Kambhampati et al., 1996)).

Traditionally, the treatments of EBL in machine learning focused heavily on the generalization phase (Minton et al., 1989). This tends to mask the essential similarities between them and the learning approaches in CSP (e.g. (Dechter, 1990; Frost & Dechter, 1994a)). Our treatment here shows that the "object generalization" is only a small variation on the basic EBL/EDB theme. In particular, the CSP approaches justifiably ignore generalization aspects since the nogoods learned in CSP are expected to be used mostly in the intra-problem scenarios, and the CSP problem constraints are given in a completely instantiated form. When we consider CSP problems where domain constraints are described in "constraint schemas", each of whose instantiations correspond to specific constraints, there is a scope for generalization. Examples of such CSP problems include CSP instances corresponding to finding k-length solutions to a planning problem (Kautz, Selman, & McAllester, 1996) (see also Section 5.3).

4.3 Selecting a Failure Explanation

In our discussion of EDB and EBL in the previous section, we did not go into the details of how a failure explanation is selected for a dead-end leaf node. Often, there are multiple explanations of failure for a dead-end node, and the explanation that is selected can have an impact on the extent of EDB, and the utility of the EBL rules learned. The most obvious explanation of failure of a dead-end node N is the set of constraints comprising N itself. In the example in Figure 9, $E(N_5)$ can thus be $x = A \wedge y = B \wedge u = C \wedge v = D \wedge w = E$. It is not hard to see that *using N as the explanation of its own failure makes EDB degenerate into chronological backtracking* (since the node N must have been affected by every decision

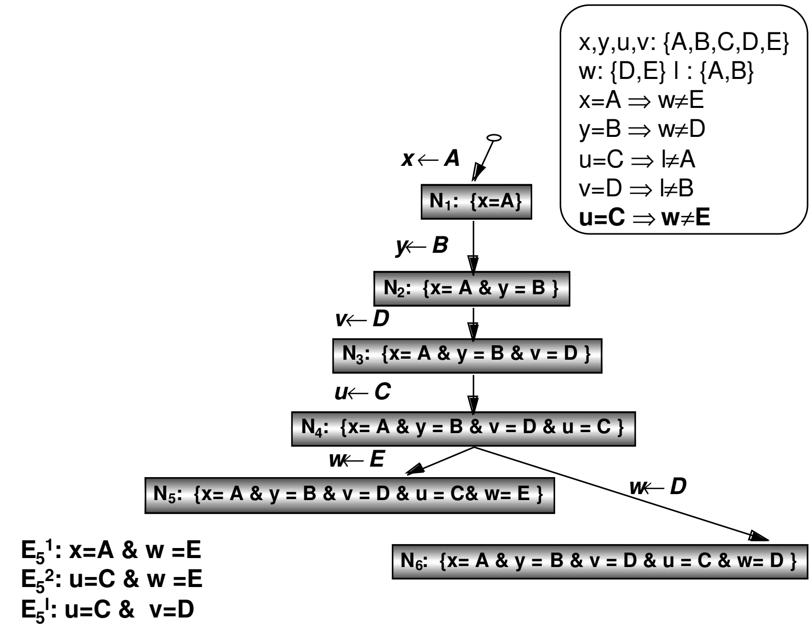

Figure 17: Example of Multiple failure explanations

that lead to it[13]). Furthermore, given the way the explanations of failure of the interior nodes are computed (see Figure 11), no ancestor N' of N can ever have an explanation of failure simpler than N' itself. Thus, no useful learning can take place.

A better approach is thus to select a smaller subset of the constraints comprising the node, which by themselves are inconsistent. In particular, we will call an explanation of failure "unbloated" if there exists a subset of the explanation that violates the same problem constraints that the original explanation is violating. The idea then is to compute unbloated explanations. For example, in CSP, if a constraint is violated by a part of the current assignment, then that part of the assignment can be taken as an explanation of failure. Similarly, the set of ordering constraints that constitute a cycle or the set of binding constraints that bind the same planning variable to two objects, can be used as starting failure explanations in planning.

Often, there may be multiple possible explanations of failure for a given node. For example, consider the modified version of the CSP problem shown in Figure 17. Here, we have a new constraint saying that $u = C \Rightarrow W \neq E$. In such a case, the node N_5 would have violated two different constraints, and would have had two failure explanations – $E_1 : x = A \wedge W = E$ and $E_2 : u = C \wedge w = E$. This brings up the question of deciding between the explanations. There are two important heuristics here:

1. Prefer explanations that are smaller in size.

2. Prefer explanations that refer to constraints that have been introduced into the node by earlier refinements.

13. We are assuming that none of the refinement decisions are degenerate; each of them add at least one new constraint to the node.

The first heuristic is best understood in terms of EBL – smaller explanations are likely to be applicable and useful in other situations, including other branches of the current search tree. and will also entail smaller match cost. The second heuristic is motivated from the EDB point of view– favoring explanations of failure that blame decisions taken earlier in the search can allow us to jump back to higher levels of the search tree.

By these heuristics, E_1 is preferable to E_2 as the explanation of failure of N_5, since E_2 would have made us backtrack only to N_2, while E_1 allows us to backtrack up to N_1. It is important to note that these are however only heuristics. It is possible to come up with scenarios where picking an explanation involving constraints introduced at lower levels could have helped more, since they combine better with the explanations regressed from other branches (about which we don't know at the time we pick the current explanation). To see this, consider a situation where we have two explanations for a failure node $E_1 : x_1 = A \wedge x_5 = C$ or $E_2 : x_2 = B \wedge x_5 = C$. Now, assuming x_1 was given value before x_2 was, the first explanation would have been preferred by the heuristic compared to the second one. Suppose x_5 can only have C or D as its values, and when trying to give $x_5 = F$, we find that we fail again, but this time because $E_3 : x_2 = B \wedge x_5 = D$ is a failure explanation. Now, had we selected E_2 at the earlier node, then E_2 and E_3 would have simplified to give $(x_2 = B)$ as the combined failure explanation. By selecting E_1, we only get the explanation: $x_1 = A \wedge x_2 = B$. which is a non-minimal explanation.[14].

Another approach is to generalize the EDB/EBL algorithm by allowing multiple explanations of failure for each of the leaf nodes. For example, we could consider $E_1 \vee E_2$ as the explanation of failure of N_5. Although there is no theoretical problem in doing this, in practice, handling the disjunctive explanations and simplifying them appropriately can be computationally expensive. It is possible that the overhead of this eager learning is not adequately offset by its benefits.[15]

In particular, if node N has two children N_1 and N_2 and N_1 has failure explanations E_1^1, E_2^1, E_3^1, and N_2 has failure explanations E_1^2, E_2^2, E_3^2, and d_1 and d_2 are the decisions leading from N to N_1 and N_2 respectively, then the failure explanation at N_1 will be a disjunction of six conjunctive explanations:

$$[d_1^{-1}(E_1^1) \wedge d_2^{-1}(E_1^2)] \vee [d_2^{-1}(E_1^1) \wedge d_2^{-1}(E_2^2)] \vee \cdots [d^{-1}(E_3^1) \wedge d^{-1}(E_3^2)]$$

While some of these can be removed based on subsumption relation (if an explanation is a subset of another explanation, the second one can be removed), we may still be left with many explanations at multiple nodes. It is not clear whether the added expense of keeping multiple explanations of failure will be offset by the savings of higher level backtracking, or smaller stored failure explanations.[16]

4.4 Cost of computing explanations

Although it is easy to recognize dead-end nodes and provide them an explanation (if not a minimal explanation) of failure in CSP, even this can be computationally expensive in

14. Similar points are raised by Minton and Etzioni (Etzioni & Minton, 1992). This phenomenon is also similar to the "bridging effect" that Prosser talks about (Prosser, 1993)
15. Dechter (Dechter, 1990) uses multiple failure explanations for leaf node failures –but she never propagates them upwards or simplifies them (since she is getting the minimal explanations to begin with). So, the implicit disjunction involved in multiple failure explanation is not a problem for her.
16. Decther (1990) considers multiple explanations of the leaf nodes. However, she does not do any regression or propagation (see Section 5.2), and thus doesn't incur the explanation handling costs described here.

tasks such as planning. Refinement planners can often go into "looping" making several locally seemingly useful but globally useless refinements (Kambhampati, 1995; Smith & Peot, 1996). Typical solutions for controlling such looping involve the use of depth limited search strategies, which initiate backtracking when a depth limit is crossed. Since there is no detectable inconsistency in the search node or(partial plan) at the depth limit, it is hard to recognize or explain dead-ends in such situations, which severely inhibits the effectiveness of EDB and EBL.

Although it is possible to provide a theory of loop-detection and pruning (Kambhampati, 1995), and use it to explain why it is sound to prune the plan, the explanations constructed in this way tend to be rather long, and are thus of limited utility in EDB and EBL. In fact, some approaches for handling EBL utility problem (see Section 4.5) explicitly prohibit learning from looping failures for this very reason (Etzioni, 1993). Ultimately, if there is a significant amount of looping, failure based approaches do not help enough in controlling the search of a planner (c.f. (Kambhampati et al., 1996)).

One idea is to find other types of failures that have smaller explanations. Part of the reason for the lack of detectable failures in planning, as contrasted to CSP problems, is a lack of global problem/domain constraints. Because of this we are stuck with looking for explicit inconsistencies among the constraints of the plan and the task level knowledge (such as ordering, binding and link inconsistencies). In CSP terms, it is like looking for inconsistencies of type "the current node gives two values to the same variable." A much richer source of failures will be violation of domain/problem constraints. As mentioned in Section 2.3, the lack of domain constraints is really an artifact of simplistic problem formulation. Situation is likely to improve when global resource and capacity constraints are made part of the problem specification, since plans suffering from looping and other undetectable failures may also violate the global constraints.

Another approach for dealing with unexplainable deadend nodes is using "partially sound" explanations of failure. This latter is motivated by the fact that although proving that a partial plan is inconsistent is hard, often we may know that the presence of a set of features is losely "indicative" of the unpromising nature of then partial plan. For example, FAILSAFE system (Bhatnagar & Mostow, 1994) constructs explanations that explicate *why the current node is not the goal node*, inspite of many refinements.

Relaxing soundness requirement on failure explanations will allow EBL to learn with incomplete explanations, thus improving the number of learning opportunities. We are currently experimenting with a variant of this approach, where such partial explanations of failure are associated with numerical certainty factors between 0 and 1 (to signify their level of soundness) (Zimmerman & Kambhampati, 1997). The explanation of failure of an interior node will have a certainty factor that depends on the certainty factors of the explanations of failure of its children nodes. Similarly, the search control rules learned from these failure explanations will also inherit the certainty factors of the explanations.

Of course, learning with unsound explanations of failure will lead EBL to learn unsound search control rules, which, if used as pruning rules, can affect the planner's completeness. We can handle this by considering such search control rules to black-list, rather than prune plan refinements.

Although sacrificing soundness seems like a rather drastic step, it should be noted that "correctness" and "utility" of a search control rule are not necessarily related. Utility is a function of the problem distribution that is actually encountered by the planner, and thus, it is possible for a rule with lower certainty factor to have higher positive impact on the efficiency than one that is correct.[17]

4.5 Remembering (and using) Learned Failure Explanations

Another issue that is left open by our EDB/EBL algorithm is how many learned failures should be stored. Early formalizations of dependency directed backtracking (e.g. (Stallman & Sussman, 1977)) have made the rather strong assumption that all failure explanations would be stored. Since there can be an exponential number of failure explanations in the worst case, the whole idea of intelligent backtracking got a bad name in some circles.[18] Specifically, there is a tradeoff in storage and matching costs on one hand and search reductions on the other. Storing the failure explanations and/or search control rules learned at all interior nodes could be very expensive from the storage and matching cost points of view. A better solution to this tradeoff is to store "some" (rather than all) failure explanations. The CSP and machine learning researchers took different approaches in deciding which nogoods to store. The differences in the approaches are to a large extent motivated by the differences in CSP and planning/problem solving tasks. The nogoods learned in CSP problems have traditionally only been used in intra-problem learning[19], to cut down search in the other branches of the same problem. In contrast, work in machine learning concentrated more on inter-problem learning. It is also interesting to note that CSP community concentrated mostly on the storage cost, while machine learning community concentrated mostly on the match cost (which becomes important given that explanations are generalized before being stored, and can thus look relevant in many situations).

Researchers in CSP (e.g. (Dechter, 1990; Tsang, 1993; Bayardo & Miranker, 1996)) concentrated on the syntactic characteristics of the nogoods, such as their size, minimality or relevance, to decide whether or not they should be stored. Specifically, Dechter and her co-workers suggested storing failure explanations that are "minimal" in that no subset of that failure explanation will entail inconsistency (with either the explicit or derived constraints). Since checking for minimality can be costly, another related idea is to store failure explanations that are smaller than a certain size bound. k-th order size based learning stores only those nogoods that involve at most k variables. Keeping k small will presumably reduce match and storage cost, while also increasing the chance that they will be useful in other search branches.

The two schemes above are static in that once a nogood is remembered, it will never be forgotten. Another class of approaches forget some of the stored nogoods as the search progresses. Jiang et. al. (Jiang, Richards, & Richards, 1994) propose forgetting previously stored nogoods that are subsumed by (i.e., are less general than) the newly learned nogoods. Another idea is to use some syntactic notion of the short-term relevance of the nogood for the current search node, to decide whether to purge some of the nogoods (which may

17. As an analogy, consider a physician who has two diagnostic rules, one that is completely certain, but is about a relatively rare disease (e.g. ebola virus syndrome), and another which has low certainty, but is about a frequently occurring disease (e.g. common cold). Clearly, the latter rule may be much more useful for the physician practising in a US city, than the latter.

18. The only use of storing all nogoods is that EDB with full learning ensures compositionality – i.e., mixture of independent subproblems can be solved in time additive in the original subproblems. However this seems like a terrible price to pay for compositionality.

19. A minor exception is the work of Verfaillie and Schiex (1994), who consider using the nogoods when the CSP problem is modified by adding and deleting constraints

eventually be re-learned again[20]). Bayardo and Miranker (Bayardo & Miranker, 1996) discuss a family of relevance based learning schemes. A k^{th}-order relevance based learning scheme keeps a nogood as long as it differs in at most k variable-value pairs from the current partial assignment. When backtracking occurs, any nogoods that differ from the new partial assignment in more than k variable-value pairs are deleted. Bayardo and Miranker present empirical studies that show that relevance-based learning schemes typically work better than sized-based learning schemes.

Researchers in machine learning concentrated instead on utility analyses that keep usage statistics on the remembered nogoods (rules) (Minton., 1990; Gratch & DeJong, 1992; Greiner, 1996). These statistics include (a) the number of times the rule was used (b) the cost of matching the rule and (c) the search reduction provided by the rule. Based on these statistics, rules deemed to be less useful are "forgotten" or pruned. Since these statistics depend upon the problems actually encountered by the problemsolver, analysis based on them can be more distribution sensitive than pure syntactic approaches for nogood storage. There do exist some machine-learning approaches that concentrate on syntactic criteria. For example, many learning systems proscribe learning and storing "recursive explanations" (Etzioni, 1993; Letovsky, 1990), as these will typically necessitate costly matching phases. Techniques such as "forgetting subsumed (less general) explanations", while applicable, tend to be too costly to implement in planning and problemsolving scenarios as these will require matching the newly learned rule against all previously learned rules.

5. Relations to existing work

Figure 18 provides a rough conceptual flow chart of the existing approaches to EDB and EBL in the context of our formalization. In this section, we briefly describe the relations between these approaches and our formalization.

5.1 Relation to existing backtracking algorithms

We start by noting that there is a lot of terminological confusion about the word "dependency directed backtracking" in CSP literature, with generic terms getting tangled up with very specific meanings. For example, as we mentioned earlier, some authors (c.f. (Stallman & Sussman, 1977; Ginsberg, 1993; Baker, 1995)) consider dependency backtracking to refer to doing EDB and storing all learned nogoods (this despite the fact that nothing in the phrase "dependency directed backtracking" hints at nogood storage!). Similarly, the term backjumping has been used originally by Gaschnig (Gaschnig, 1977) to refer to the act of backtracking intelligently from the leaf nodes alone–in other words, there was no propagation, and computation of interior node failure explanations. However, some recent descriptions use the term to refer to the process of propagation and interior node failure explanation.

In addition to the terminological differences, the descriptions of CSP backtracking/learning approaches may look different from the formalization here for two reasons. First, most work on traditional CSP has concentrated on "binary constraint satisfaction problems" (BCSPs)– where all the constraints are between pairs of variables. BCSPs admit a specialized datastructure called "constraint graphs"–which contain the variables as the

20. This is sort of like the "purge the least recently used page" strategy used in page-caching schemes in operating systems.

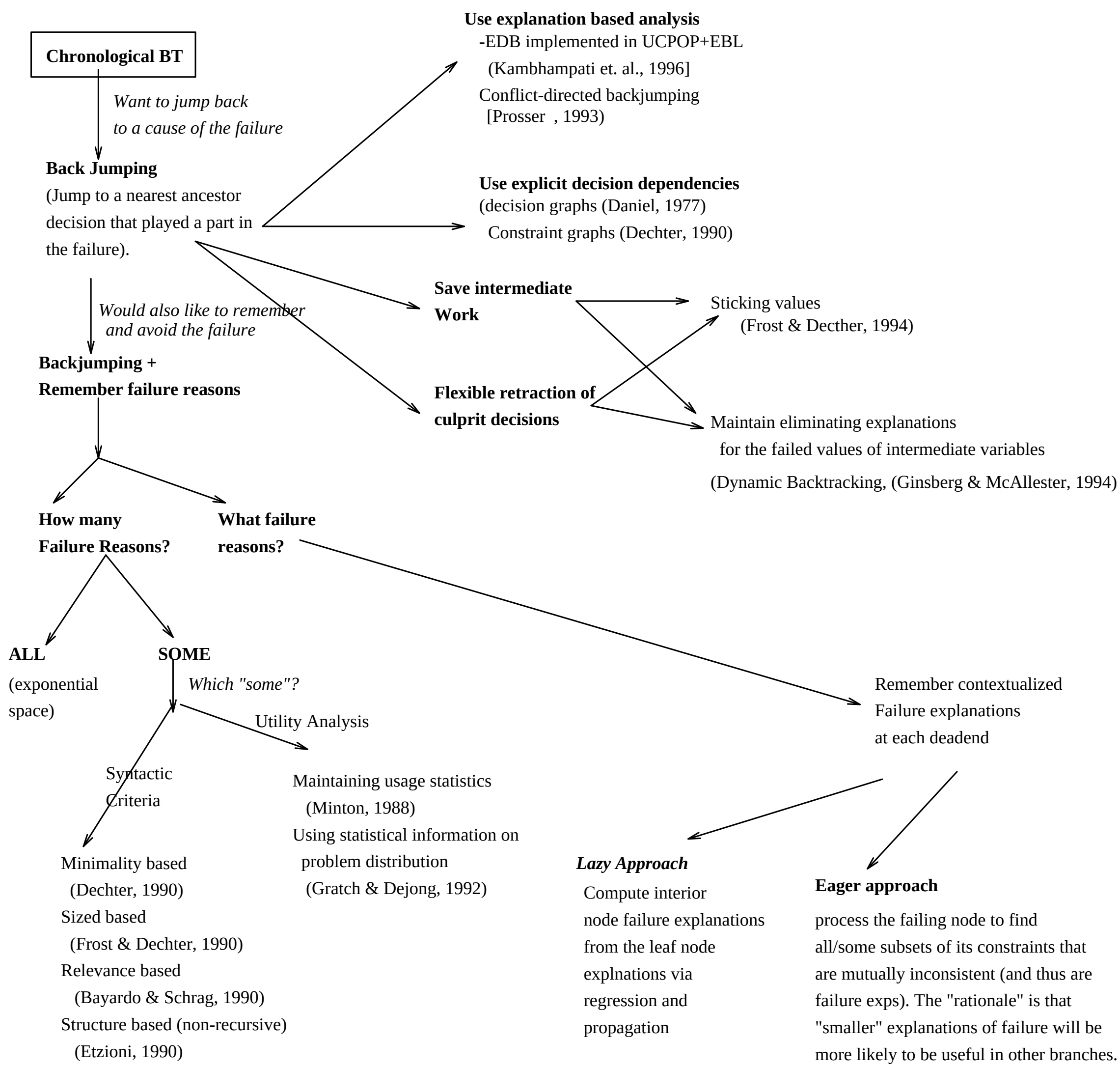

Figure 18: A schematic flow chart tracing the connections between implemented approaches to EDB and EBL

vertices and an edge between two vertices if there is a constraint relating the corresponding variables. BCSPs also accommodate a specialized representation for explanations of failure. Since all constraints are binary, the failure caused when we assign a value to a variable can be blamed squarely on a single other assigned variable in the current partial assignment. Finally, most CSP techniques do not explicitly talk about regression as a part of backtracking. This is because in CSP there is a direct one-to-one correspondence between the current partial assignment in a search node and the decisions responsible for each component of the partial assignment.[21] For example, a constraint $x = a$ must have been added by the

21. This is not strictly true in algorithms that interleave forward checking and refinement, since the forward checking phase may infer many assignments at once. However, even here, we can keep inferred assignments separate from the assignments added by refinement decisions, and erase all the inferences as soon as we backtrack over the last refinement assignment that allowed the inferred constraint.

decision $x \leftarrow a$. Thus, in the example in Figure 13 it would have been easy enough to see that we can "jump back" to N_2 as soon as we computed the failure explanation at N_4.

This special structure of BCSP problems has facilitated specialized versions of EDB algorithm such as "Graph-based back jumping" (Dechter, 1990) that use the constraint graphs to help in deciding which decision to backtrack to. In planning, no one-to-one correspondence exists between decisions and the constraints in the node. For example, both demotion and step-addition decisions may add orderings to a plan, and step-addition may add a variety of constraints, including orderings, steps and auxiliary constraints. Even here, it is possible to build more complex dependency structures. An example of such syntactic structures for planning are the "decision graphs" (Daniel, 1977). One way of thinking about constraint graphs and decision graphs is to see them as providing some partial details about the explanation of failure, without requiring an explicit failure analysis. The disadvantage is that by not considering and reasoning with failure explanations explicitly, these techniques typically are incapable of using propagation techniques to compute interior node failure explanations, and to support learning.

In a way, constraint graphs and decision graphs attempt to solve the same problem that is solved by regression. However, the semantics of these structures are often problem dependent, and storing and maintaining them can be quite complex (Petrie, 1992). In contrast, the notion of regression and propagation is problem independent and explicates the dependencies between decisions on an as-needed basis. On the other hand, regression and propagation work only when we have a declarative representation of decisions and failure explanations, while dependency graphs may be constructed through procedural or semi-automatic means.

The CSP backtracking idea that is closest to our EDB formalization is the "conflict directed backjumping" (CBJ) approach proposed by Prosser (Prosser, 1993). This algorithm is originally proposed for binary CSP problems. When a new variable x is given a value v in the context of the current partial assignment N, CBJ checks to see if any of the constraints involving x and some variable y that has an assignment in N is violated (recall that the constraints are all binary). If a violated constraint is found, then the corresponding variable y is added to the conflict set of the variable v, and a new value is tried for v. If all the values of v are found to be inconsistent, then CBJ picks the most recently instantiated variable j from the conflict set of v and backtracks to it. At this point, the conflict set of v, minus the variable j, is added to the conflict set of j (Prosser calls this step "conflict set merging"). If j still has unexplored values, one of them is assigned to j, and the search continues. If j has no more unexplored values, we continue backtracking–this time to the most recently assigned variable that appears in j's conflict set.

Although the description of the whole algorithm is in terms of conflict sets and their values, it is easy to see that conflict sets are really a stylized representation of the explanations of failure for BCSP problems. The process of "conflict set merging" essentially computes the interior node failure explanation, and corresponds to what we have called the "propagation" process (in particular, note that our propagate procedure, in Figure 7, line 3.1.2 incrementally accumulates the explanation of failure of interior nodes). The cascade of backtracks facilitated by CBJ are similar to recursive propagation (line 3.2 in Figure 7). The regression process is not explicitly considered in CBJ because of the one-to-one correspondence between decisions and the constraints they add, and the addition of flaw description to interior node failure explanation is not done in CBJ for reasons discussed in Section 4.1. Thus, CBJ can be seen as an instantiation of EDB procedure for the special case of BCSPs[22]

The NR_* approach, proposed by Schiex and Verfaillie (1993) at about the same time as the CBJ algorithm combines EDB and a generalization of EBL in a single algorithm. One important difference is that NR_* allows resolving learned explanations of failure to generate explanations of failure independent of the propagation stage (see Section 5.2). Thus, sometimes it may allow backtracking to higher levels than CBJ (and EDB), at the expense of increased storage and costlier processing of learned explanations (also see next section).

In the context of planning, the backtracking scheme used by UCPOP+EBL (Kambhampati et al., 1996) is essentially identical to EDB. Empirical results presented in (Kambhampati et al., 1996) demonstrate that this backtracking scheme improves the performance of the planner significantly. To our knowledge, UCPOP+EBL is the only planner to have used an explanation directed backtracking scheme. In fact, it is in the course of our work on UCPOP+EBL that I became interested in the close relation between EDB and EBL ideas and the many variations they take on in planning and CSP literature.

The connections between dynamic backtracking algorithm (Bruynooghe, 1981; Ginsberg, 1993; Ginsberg & McAllester, 1994) and the EDB framework has already been explained in Section A. The discussion there clarifies the claims about dynamic backtracking –including polynomial stored nogoods, saving intermediate work, and backtracking to an earlier variable.

5.2 EBL as a lazy way of learning induced nogoods

As long as the leaf nodes are given sound explanations of failure, the interior node failure explanations learned by the EBL process are "induced (implicit) constraints" in that they can be deduced logically from the explicitly specified problem and domain constraints. Clearly, EBL is not the only way for deriving induced constraints – any logical deduction mechanism operating on the explicit knowledge can derive the constraints (and control rules) derived by EBL. There are a variety of such approaches, all of which can be characterized as being "more eager" in deriving the implicit constraints.

Perhaps the most eager approach for learning induced constraints is to do undirected or "forward" deduction on the domain/problem knowledge. Examples of this type of learning include "partial evaluation" techniques used in program optimization (van Harmelen & Bundy, 1988), and the constraint propagation (local consistency enforcement) techniques used in CSP (Tsang, 1993). The utility problem that we discussed in the context of EBL (Section 4.5) applies equally well to the constraints derived by these direct-inferencing approaches. In addition, since uncontrolled deduction can be computationally expensive, direct-inferencing approaches also have to worry about controlling the amount of computation spent in the inference. Normally various syntactic criteria are used to affect this control. For example, in the case of constraint propagation in CSP, the degree of inference is measured by the level of consistency enforcement. Enforcement of strong k-level consistency takes $O(n^k)$ time, and essentially makes explicit all constraints of size $\leq k$.

22. Kondrak and vanBeek (1995) point out that the completeness of CBJ has never been formally proven, and provide a proof in terms of their framework. Seeing CBJ as an instance of EDB provides an alternative proof of completeness and correctness of CBJ.

The primary difference between EBL and these direct inference approaches is that EBL approaches tend to be "example driven". Specifically, they wait for a failure to occur, and only then learn (deduce) implicit constraints that are relevant to that failure. The assumption is that such failures are more likely to recur, thus making the implicit constraints worth storing explicitly. There are various degrees of discernible eagerness even within the "example directed" learning schemes. The traditional EBL framework that I described in Section 3.5 is lazy in that it will start with leaf node failure explanations and combine them only to derive ancestor node failure explanations. Thus, two explanations of failure will be combined only when they occur as the failure explanations of two sibling nodes in the current search tree. Traditional EBL does not consider direct resolution of stored explanations. In contrast, some approaches, such as the NR_* (Schiex & Verfaille, 1994; Schiex & Verfaillie, 1993) allow combining ("resolving") any set of stored failure explanations. They can thus derive more implicit constraints than traditional EBL.[23] Dechter (Dechter, 1990), describes an even more eager approach – which does nogood learning by just analyzing the failing leaf nodes, without reasoning about interior nodes. In particular, the approach enumerates all failure explanations of the node that violate either explicitly stated constraints or implied constraints. It is interesting to note that since the constraint sets of interior nodes are subsets of the constraint sets of leaf nodes, when we enumerate all possible failure explanations of the leaf nodes, we also implicitly enumerate the failure explanations of the interior nodes. For example, in Figure 12, the explanation of failure of the interior node N_4 also holds true in the leaf node N_5 (and N_6), and thus could in principle have been isolated just by looking at N_5. Thus, in theory we can avoid regression and propagation procedures all together (Dechter, 1990).

While all the approaches are explicating the implicit constraints, the tradeoffs between eager and lazy (or "example-directed") approaches are related to the utility problem. Specifically, a possible advantage of computing interior node failure explanations in the EBL way is that the regression and propagation procedures compute failure explanations based on the search tree that is actually generated by the problem solver, it is possible that the failure explanations generated by this process are more *utile*, in that they have a higher chance of being applicable in other parts of the search tree or in other problems (see Section 4.5).[24]

Of course, being completely example-driven has its drawbacks too. Etzioni and Minton (Etzioni & Minton, 1992) argue that often EBL produces overly-specific knowledge that leads to inefficient inter-problem transfer, precisely because it is guided purely by examples. They suggest using hybrid techniques that use both direct inferencing and example driven learning to improve the generality of learned knowledge.

Within CSP literature, there is overwhelming evidence that complementing EBL and EDB with low level consistency enforcement in fact leads to the best performance. This idea combines the strength of direct inferencing and example guided learning techniques: low-order constraint propagation effectively makes explicit lower-order failure explanations (typically involving pairs of variables) in a low-order polynomial time. This makes search encounter failures less often. When the failures are encountered, they will be higher-order

23. Schiex and Verfaillie (1994) also discuss an even eager example guided learning method called "Stubborn learning" which involves continuing to refine a node even after a failure has been discovered in it, to experience and learn more failures. This somewhat quixotic method seems to have only had a partial empirical success.

24. the situation here is similar to the tradeoffs between complete regression vs. example guided regression; see Section 3.2.

ones, and these are explicated by EBL analysis. In fact, most winning CSP search algorithms combine "forward checking," which does 2-level consistency enforcement with respect to the current partial assignment, with EBL and EDB algorithms (Frost & Dechter, 1994c; Bayardo & Schrag, 1997).

5.3 Posing Planning as CSP

Although we talked about EDB/EBL ideas in planning and constraint satisfaction separately, there is another distinct body of research that attempts to pose planning problems directly as CSP problems. The complete planning problem cannot be posed as a CSP problem since the former is P-Space complete while the latter is NP-complete. However, it is possible to pose subparts of the planning problem as CSP problems. In particular, the problem of "finding if any of the minimal candidates (linearizations) of a set of partial plans corresponds to a solution" can be posed as a CSP problem (Kambhampati & Yang, 1996). Yang's WATPLAN (Yang, 1992) and Kambhampati and Yang's (1996) UCPOP-D pose the problem of checking a single plans linearizations for solutions as a CSP, while the more recent research efforts including Graphplan (Blum & Furst, 1995) and SATPLAN (Kautz & Selman, 1996), and Descartes (Joslin & Pollack, 1997) encode the problem of sorting through the linearizations of a large set of partial plans (represented in a disjunctive fashion) as a CSP. In all these cases, the usual search tradeoffs in CSP apply (Frost & Dechter, 1994c). For example, the CSP (solution extraction) phase of Graphplan (Blum & Furst, 1995) uses a combination of constraint propagation (propagation of 2-sized mutex constraints) and a form of EBL (memoizing higher-order mutex constraints learned through search failures) to improve performance.

6. Summary and Conclusion

In this paper, I formally characterized two long standing ideas – dependency directed backtracking and explanation based learning – in the general task-independent framework of refinement search. I showed that at the heart of both is a process of explaining failures at leaf nodes of a search tree, and regressing them through the refinement decisions to compute failure explanations at interior nodes. Backtracking involves using the computed failure explanation to decide which decision point to go back to, and EBL involves storing and applying failure explanations of the interior nodes in other branches of the search tree or other problems.

This task-independent characterization of EBL and DDB, coupled with the fact that planning and CSP tasks can be modeled in terms of refinement search, helps us compare and understand the tradeoffs offered by a multitude of backtracking and learning techniques developed independently for planning and CSP. My analysis shows that most of the differences between CSP and planning approaches to EBL and DDB revolve around different solutions to: (a) How the failure explanations are selected (b) How they are contextualized (which involves deciding whether or not to keep the flaw description and the description of the violated problem constraints) and (c) How the storage of explanations is managed. The differences themselves can be understood in terms of the differences between planning and CSP problems as instantiations of refinement search.

I have also provided a comprehensive discussion of related work, showing how the unified view covers and clarifies the existing algorithms. This discussion also shows how ideas behind dynamic backtracking emerge as extensions of EDB, and explains the relations between notions of constraint propagation and EBL.

I believe that the insights gained from my unified treatment of EBL and EDB in CSP and Planning facilitates a significantly greater cross-fertilization of ideas among these communities. I speculate on some of these below. Please note that this is not meant to be an exhaustive list, but is meant to be indicative of the types of cross-fertilization of ideas that might be supported by this paper.

Inter-problem learning in CSPs: As we noted, CSP has always been concerned about intra-problem learning without generalization. This makes sense under the classical CSP assumption that every problem is different from every other problem – a problem comes with its own fresh set of variables and constraints, and all of them are treated individually. This assumption is too pessimistic however when real world tasks are modeled directly as CSP instances – the problems will share several of the variables as well as the several of the domain-wide constraints. For example, suppose we model a jobshop scheduling problem in CSP. The capacity constraints of the plant as well as the machines in the plan are not likely to change from problem to problem. Furthermore, traditional CSPs start with completely instantiated constraints. In many problems, we can see the individual constraints to be instantiations of specific constraint schemas, obtained by substituting specific object names into the schema (c.f. (Kautz et al., 1996)). In such cases, there is going to be a large amount of shared structure between problems making inter-problem learning as well as generalization very attractive. It would be interesting to see how the utility analysis techniques from EBL in Planning can be modified to fit these requirements.

Using global constraints to do constraint propagation and failure detection in planning: We have noted that the best CSP search techniques combine EBL and EDB with low-degree constraint propagation. Since planning can also be cast as a refinement search, it is reasonable to expect that similar techniques work well for planning too. As we mentioned, to our knowledge, the only planner that uses EDB techniques is UCPOP+EBL (Kambhampati et al., 1996). It is interesting to speculate as to why such techniques have not been widely used in planning literature.[25] Part of the problem, as we pointed out in Sections 2.3 and 4.4, is that planning problems do not contain enough global constraints with respect to which inconsistencies can be detected, and constraint propagation can be done. The conventional wisdom behind separating resource and capacity constraints out of the planning is that satisfying those constraints is best seen as a "scheduling" activity. The idea is to use planning techniques only to come up with feasible action sequences, to which resources are assigned in the scheduling phase. Although this looks like a good divide-and-conquer technique, I believe that it may actually be counter-productive for the planning efficiency. By removing resource and capacity constraints from planning, we make the problem artificially under-specified, making the search harder. By keeping the constraints up-front, we can use them to bias the search of the planner (for example by propagating those constraints through the current partial plan), and also use them to explain the failure of unpromising plans (for example, most looping plans may wind up violating resource

25. Here, we are not considering the use of constraint propagation and EBL in planners like WATPLAN (Yang, 1992) and Graphplan (Blum & Furst, 1995) since as we discussed in Section 5.3, since in these planners, the techniques are employed in the solution extraction phase, which is cast as a CSP problem directly. We are interested in the direct use of these techniques in the refinement of partial plans.

constraints). This argues for richer problem specifications that have hither-to been shunned in classical planning.

As we noted, another important difference between planning and CSP is that the description of search nodes in CSP (partial assignments) is much simpler than that in planning (partial plans). This facilitates a very simple representation of the failure explanations for CSP making a variety of backtracking algorithms feasible. Although it is easy to adapt EDB/EBL to the more complex partial plan representations, ideas such as dynamic backtracking become harder (since eliminating explanations will contain a variety of constraints). This suggests planning search spaces that have more uniform representations are perhaps more amenable to backtrack techniques. One idea would be to "compile" plan representations down to a more uniform language before applying EDB and EBL. State-variable based state representations provide a promising avenue (Muscettola et al., 1991).

Acknowledgements

The ideas described here developed over the course of my interactions with Suresh Katukam, Gopi Bulusu and Yong Qu. I thank them for them for their insights. I also thank Suresh Katukam and Terry Zimmerman for their critical comments on a previous draft, and Steve Minton for his encouragement on this line of work. A preliminary version of this paper was presented at AAAI-96 (Kambhampati, 1996). The comments of the JAIR reviewers, as well as those of Roberto Bayardo were helpful in revising the paper. This research is supported in part by NSF research initiation award (RIA) IRI-9210997, NSF young investigator award (NYI) IRI-9457634 an ARPA/Rome Laboratory planning initiative grants F30602-93-C-0039 and F30602-95-C-0247, and an ARPA AASERT grant DAAH04-96-1-0231.

References

Baker, A. (1995). *Intelligent backtracking on constraint satisfaction problems: Experimental and Theoretical results.* Ph.D. thesis, University of Oregon.

Bayardo, R., & Miranker, D. (1996). A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proc. AAAI-96.*

Bayardo, R., & Schrag, R. (1996). Using csp look-back techniques to solve exceptionally hard sat instances. In *Ppls of Constraint Programming Languages (lecture notes in CS, v. 1118).*

Bayardo, R., & Schrag, R. (1997). Using csp look-back techniques to solve real-world sat instances. In *Proc. AAAI-97 (to appear).*

Bhatnagar, N., & Mostow, J. (1994). On-line learning from search failures. *Machine Learning*, *15*, 69–117.

Blum, A., & Furst, M. (1995). Fast planning through plan-graph analysis. In *Proc. IJCAI-95.*

Bruynooghe, M. (1981). Solving combinatorial search problems by intelligent backtracking. *Information processing letters*, *12*, 36–39.

Daniel, L. (1977). Planning: Modifying non-linear plans. Tech. rep. DAI Working Paper: 24, *University Of Edinburgh,*.

Dechter, R. (1990). Enhancement schemes for learning: Back-jumping, learning and cutset decomposition. *Artificial Intelligence*, *41*, 273–312.

Dechter, R., & Dechter, A. (1988). Belief maintainance in dynamic constraint networks. In *Proc. AAAI-88*.

DeJong, G. (1996). *The Computer Science and Engineering Handbook*, chap. 21. Explanation-based learning. CRC Press.

Estlin, T., & Mooney, R. (1996). Multi-strategy learning for search control for partial order planning. In *Proc. AAAI-96*.

Etzioni, O. (1993). A structural theory of explanation-based learning. *Artificial Intelligence*, *60(1)*.

Etzioni, O., & Minton, S. (1992). Why ebl produces overly-specific knowledge: A critique of the prodigy approaches. In *Proc. Machine Learning Conference*.

Frost, D., & Dechter, R. (1994a). Dead-end driven learning. In *Proc. AAAI-94*.

Frost, D., & Dechter, R. (1994b). In search of best constraint satisfaction search. In *Proc. AAAI-94*.

Frost, D., & Dechter, R. (1994c). In search of the best constraint satisfactions earch. In *Proc. AAAI-94*.

Gaschnig, J. (1977). A general backtrack algorithm tha teliminates most redundant tests. In *Proc. IJCAI-77*.

Ginsberg, M. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research*, *1*, 25–46.

Ginsberg, M., & McAllester, D. (1994). Gsat and dynamic backtracking. In *Proc. KRR*.

Gratch, J., & DeJong, G. (1992). Composer: A probabilistic solution to the utility problem in speed-up learning. In *Proc. AAAI-92*, pp. 235–240.

Greiner, R. (1996). Palo: a probabilistic hill-climbing algorithm. *Artificial Intelligence*, *84*, 177–208.

Ihrig, L., & Kambhampati, S. (1996). Design and implementation of a derivational replay system based on a partial order planner. In *Proc. AAAI-96*.

Jiang, Y., Richards, T., & Richards, B. (1994). No-good backmarking with min-conflict repair in constraint satisfaction and optimization. In *Proc. 2nd Principles and Practice of Constraint Programming workshop*.

Joslin, D., & Pollack, M. (1997). Is least commitment always a good idea?. In *Proc. AAAI-96*.

Kambhampati, S. (1995). Admissible pruning strategies for plan-space planners. In *Proc. IJCAI-95*.

Kambhampati, S. (1996). Formalizing dependency directed backtracking and explanation-based learning in refinement search. In *Proceedings of AAAI-96*.

Kambhampati, S., Katukam, S., & Qu, Y. (1996). Failure driven dynamic search control for partial order planners: An explanation-based approach. *Artificial Intelligence*, *88*.

Kambhampati, S., Knoblock, C., & Yang, Q. (1995). Planning as refinement search: A unified framework for evaluating design tradeoffs in partial order planning. *Artificial Intelligence (special issue on Planning and Scheduling)*, *76*, 167–238.

Kambhampati, S., & Srivastava, B. (1995). Universal classical planner: An algorithm for unifying state-space and plan-space planning. In *Proc. 3rd European Workshop on Planning Systems*.

Kambhampati, S., & Yang, X. (1996). Role of disjunctive representations and constraint propagation in planning. In *Proc. KR-96*.

Kautz, H., & Selman, B. (1996). Pushing the envelope: Plannng, propositional logic and stochastic search. In *Proc. AAAI-96*.

Kautz, H., Selman, B., & McAllester, D. (1996). Encoding plans in propositional logic. In *Proc. KR-96*.

Kondrak, G., & vanBeek, P. (1995). A theoretical evaluation of selected backtracking algorithms. In *Proc. IJCAI-95*.

Letovsky, S. (1990). Operationality criteria for recursive predicates. In *Proc. AAAI-90*.

McAllester, D., & Rosenblitt, D. (1991). Systematic nonlinear planning. In *Proc. 9th AAAI*.

Minton., S. (1990). Quantitative results concerning the utility of explanation based learning. *Artificial Intelligence*, *42*, 363–391.

Minton, S., Carbonell, J., Knoblock, C., Kuokka, D., Etzioni, O., & Gil, Y. (1989). Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, *40*, 63–118.

Mittal, S., & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. In *Proc. AAAI-90*.

Munoz-Avila, H., & Weberskirsch, F. (1996). Planning for manufacturing workpieces by storing, indexing and replaying planning decisions. In *Proc. 3rd Intl. Conference on AI Planning Systems*.

Muscettola, N., Smith, S., Cesta, A., & D'Aloisi, D. (1991). Coordinating space telescope operations in an integrated planning and scheduling architecture. In *Proc. IEEE Intl. Conf. on Robotics and Automation*.

Nilsson, N. (1980). *Principles of Artificial Intelligence*. Tioga press, Palo Alto.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley.

Petrie, C. (1992). Constrained decision revision. In *Proc. 10th AAAI.*

Prosser, P. (1993). Domain filtering can degrade intelligent backtracking search. In *Proc. IJCAI-93.*

Schiex, T., & Verfaille, G. (1994). Stubbornness: a posible enhancement for backjumping and nogood recordings. In *Proc. 11th ECAI.*

Schiex, T., & Verfaillie, G. (1993). Nogood recording for static and dynamic constraint satisfaction problems. In *Proc. 5th intl. conference on tools with artificial intelligence.*

Selman, B., Levesque, H., & Mitchel, D. (1992). Gsat: A new method for solving hard satisfiability problems. In *In Proc. AAAI-92.*

Smith, D., & Peot, M. (1996). Suspending recursion in planning. In *Proc. 3rd Intl. AI Planning Systems Conference.*

Stallman, R., & Sussman, G. (1977). Forward reasoning and dependency-directed backtracking in a system for computer aided circuit analysis. *Artificial Intelligence*, *9*, 135–196.

Tsang, E. (1993). *Foundations of Constraint Satisfaction.* Academic Press, San Diego, California.

van Harmelen, F., & Bundy, A. (1988). Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, *36*, 401–412.

Verfaillie, G., & Schiex, T. (1994). Solution reuse in dynamic constraint satisfaction problems. In *Proc. AAAI-94.*

Yang, Q. (1992). A theory of conflict resolution in planning. *Artificial Intelligence*, *58*, 361–392.

Zimmerman, T., & Kambhampati, S. (1997). Using unsound failure explanations to imrprove the effectiveness of ebl. In preparation.

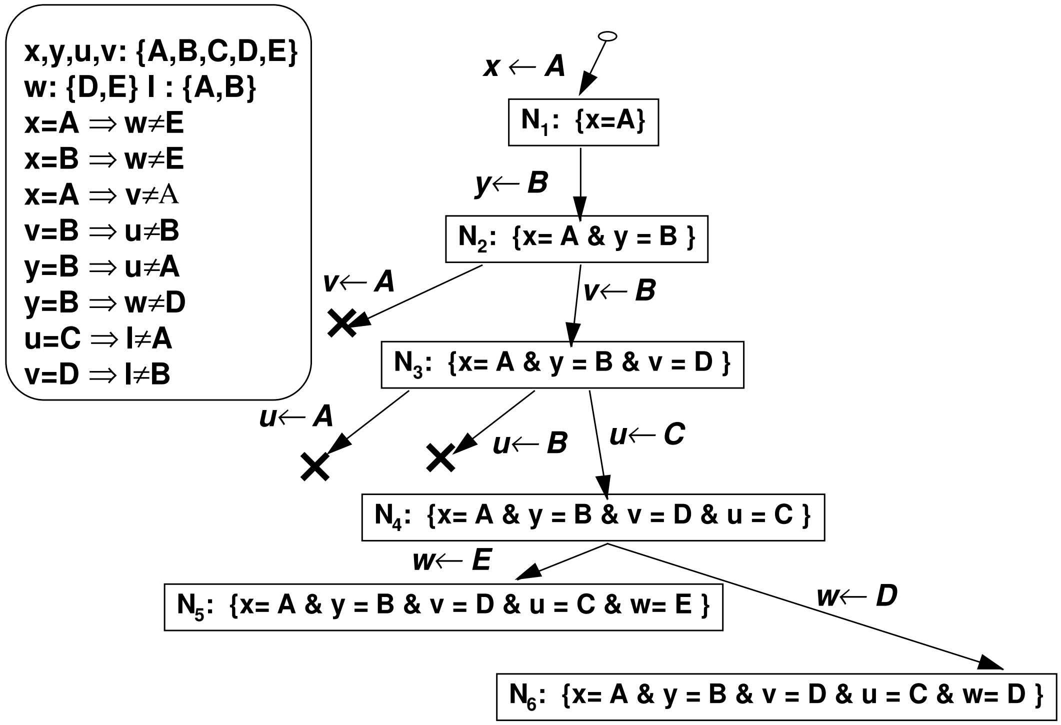

Figure 19: CSP example for illustrating the ideas of sticky values and dynamic backtracking

Appendix A. Extending the Basic EDB Framework

The EDB framework that we described in the previous sections suffers from two possible drawbacks. In this section, we will describe these drawbacks, and ways of extending EDB to overcome them.

When EDB intelligently backtracks to an ancestor node, it erases all the progress it made on the nodes between that ancestor node and the failure node, potentially wasting a lot of useful work. In particular, it is possible that we tried a variety of decisions for the intermediate flaws before settling on the ones that we just erased! Consider a variation on our CSP example, shown in Figure 19. Compared to the example in Figure 13, this one has more constraints and represents a snapshot of the search process in which the search process has already encountered several failures (shown by branches terminated with cross signs). Suppose the search is currently at the point where it has computed the explanation of failure of node N_4 to be $x = A \wedge y = B$ (having seen the failures at N_5 and N_6). If we apply EDB analysis, we will backtrack over nodes N_3 and N_4 to the node N_2. When we do this, we lose the assignment of values to variables v and u, which as far as we are concerned, have nothing to do with the failure.

A related, but less obvious, problem is the rigidness imposed by the search tree during backtracking. Specifically, the EDB algorithms backtrack only up to the last decision that has played a part in causing the failure (i.e., the last culprit decision). Sometimes, it may be useful to undo an earlier decision affecting the failure explanation, while keeping the later ones intact. In the example in Figure 19, when we fail at node N_5, and compute its explanation of failure as $x = A \wedge w = E$, we may have liked to backtrack all the way to the decision $x \leftarrow A$, while keeping the decision $w \leftarrow E$ untouched.[26]

Both the ability to keep intermediate work and to backtrack to an earlier decision taking part in an explanation of failure, require similar sorts of extensions to EDB. We will first discuss a partial solution that involves rearranging the order in which resolution possibilities are tried for intermediate flaws, and then a more complete solution that keeps justifications for each of the "tried-but-failed" decisions for the intermediate flaws.

A.1 Caching intermediate work with the use of "Sticky values"

One partial solution is to remember the current preferred resolution decisions while backtracking over flaws to higher levels, and try those decisions first when re-considering those intermediate flaws. In Figure 19, when EDB backtracks over N_4, we could remember that the preferred decision for handling the *needsAssignment* flaw corresponding to u is $u \leftarrow C$. Similarly, we can remember the preferred decision for the v as $v \leftarrow B$. Once we backtrack up to N_1, and try a different value for y, we can immediately consider the flaws corresponding to v and u and apply the preferred decisions first. Notice that we are not changing the resolution possibilities ("live domains") of u and v, but just reordering them. Thus, the completeness of the search is not affected. Within CSP, this idea is attributed to Frost and Dechter, who call it the "*sticking values*" approach. In (Frost & Dechter, 1994b), they report empirical studies showing that sticking values in conjunction with EDB can improve performance significantly.

The sticking values only remember the decision that was used for the intermediate flaws. But, a potentially more useful source of information is the set of decisions that were tried and found to fail for the intermediate flaws. For example, in Figure 19, suppose the values A and B were tried for u and A, B, C were tried for v and were found to fail, before we settled for the current values. We might want to remember this information since otherwise, we are likely to repeat the failing values once the sticking values $v = B$ and $u = C$ don't work. The obvious idea of pruning the failing values from the domains of v and u will not work since the the values may have failed because of the particular values assigned to the past variables, x and y. The failure reasons may not hold once these past variables are reassigned during backtracking. It is possible to extend the sticky values idea in an interesting way to cover this situation – rearrange the resolution decisions for each flaw in such a way that the resolution possibilities that were tried and found to fail are put towards the back of the list, with the current decision and the as yet untried decisions in the front of the list. In CSP terms, we rearrange the domains of the variables such that the values that were tried and were found to fail are kept behind the current and untried values. This will ensure that failing decisions are not tried until and unless unexplored decisions have been considered first. Since we are only re-arranging the domains, the completeness of the search is not effected. This idea is equally applicable to planning. As far as I know this generalized sticking values idea has not been tried either in planning or CSP.

26. The need for changing past variable order is not very clear when failures are incrementally detected after every refinement. We shall see the use of this capability when the search is organized as a traversal through a series of complete assignments.

A.2 Caching the justifications for eliminated decisions (Dynamic Backtracking)

Both the sticking values idea and the generalization we discussed above are only partial solutions for saving intermediate work, and supporting backtracking to earlier variables. To see this, note that black-listing previously failed resolution decisions for a flaw, when backtracking over that flaw, is not always going to be a good idea. In particular, the reason for the failure of a particular decision may not hold once we backtrack and change the way an earlier flaw has been resolved. In the example shown in Figure 19, the value $u = A$ will fail only if y is assigned the value B. If this value is backtracked over, the failure will no longer occur, and thus black-listing $u = A$ may be counter productive. A more complete solution thus involves maintaining, for each of the failing resolution possibilities of a flaw, a justification as to why it is failing. The justification can be provided by the explanation of failure of the node where the failing resolution possibilities were tried. In general, keeping track of such explanations and reasoning with them could be quite cumbersome especially in tasks such as planning where the node descriptions contain a variety of constraints (see Section 6). However, the simplicity of node and constraint representation in CSP problems makes it feasible. (We thus will restrict our attention to CSP for most of the remainder of this section.)

In the case of our running example, we need to maintain, and check, the validity of justifications for eliminating certain values of u and v. Specifically, the explanation of failure that eliminated $u = A$, viz., $y = B \wedge u = A$, can be remembered as

$$y = B \Rightarrow u \neq A,$$

with the operational interpretation that as long as $y = B$, u cannot have the value A. We shall call this an "eliminating explanation." When search is resumed after backtracking, the intermediate variables are given a value that is not eliminated by a failure explanation that is still valid (after the backtrack variable has been re-assigned). This is the general idea behind what Bruynooghe (1981) called "Intelligent backtracking," and Ginsberg and McAllester (1994) call "dynamic backtracking."[27]

There are two important issues in implementing this idea. The first is that if we remember the explanations for all the variable-value combinations that were ever eliminated, we might wind up storing an exponential number of explanations. One solution is to store only those explanations that are still relevant to the current (partial) assignment. In the example in Figure 19, as soon as we change the assignment $y = B$, we will delete the explanation "$y = B \Rightarrow u \neq A$." This way, we will keep only a polynomial number of eliminating explanations (specifically, $O(nv)$ explanations where n is the number of variables, and v is the largest variable domain size). Notice that it is possible that sometime in the future, we reassign x, reconsider the value $y = B$, and thus re-detect that the value $u = A$ will not work. This is the price we pay for keeping only the ruled-out explanations that are relevant.

The second issue is the role of search tree in the backtracking scheme. Keeping just the relevant eliminating explanations also allows us to reconstruct sufficient information

27. Although the main ideas of dynamic backtracking – including incremental maintenance of eliminating explanations, removing irrelevant eliminating explanations to keep storage polynomial, saving intermediate work, and backtracking without search tree – were all introduced (in an exceedingly brief note) by Bruynooghe (Bruynooghe, 1981) for CSP problems even before the simpler DDB algorithms (Prosser, 1993), the algorithm seems to have been forgotten for over twelve years, when it was apparently rediscovered by Ginsberg (Ginsberg, 1993).

about the search tree structure. Eliminating explanations in effect tell us the unexplored values ("live domains") for all variables. Once we keep track of the current live domains of the variables in terms of eliminating explanations in each search node (in addition to the current partial assignment), we do not need the search tree to traverse the search space in a systematic fashion. In fact, eliminating explanations lift some of the rigidness imposed by search tree on the traversal order. The main restriction imposed by a search tree is that backtracking should be done in the order the flaws were originally selected for resolution. The eliminating explanations effectively allow us to change the decision higher up in the tree, while appropriately maintaining the valid decisions for the lower level flaws, thus lifting this restriction.

Let us see how we can use the eliminating explanations to traverse the search space, starting from the failure at N_5 in Figure 19:

$$N_5 : x = A \wedge y = B \wedge v = B \wedge u = C \wedge w = E$$

Assuming that we already eliminated the value A for v and A and B for u (as shown in Figure 19, our current list of eliminating explanations will be:

$$\begin{gathered} \mathit{Eliminating\ Exp} : y = B \Rightarrow v \neq A; \quad y = B \Rightarrow u \neq A; \\ v = B \Rightarrow u \neq B \end{gathered}$$

We consider the explanation of failure $x = A \wedge w = E$. Suppose, we decide to handle this failure by modifying the value of w. Once we pick the variable we want to change, we must convert the failure explanation into a directional form that eliminates the current value of the chosen variable. In our example, we rewrite the explanation $x = A \wedge w = E$ as $x = A \Rightarrow w \neq E$ (meaning that as long as the assignment contains $x = A$, w cannot be equal to E) and keep this as one of the eliminating explanations of the search node. When we do this, *we effectively make x a past variable with respect to w, without fixing the relation of the other variables.* We can then change the assignment of w to any non-eliminated value. In this case we have a single possibility, $w = D$. The current search node is thus:

$$\begin{gathered} N' : \quad \mathit{Assignment} : x = A, y = B, v = B, u = C, w = D \\ \mathit{Eliminating\ Exp} : x = A \Rightarrow w \neq E; y = B \Rightarrow v \neq A \\ v = B \Rightarrow u \neq B; y = B \Rightarrow u \neq A \end{gathered}$$

At this point, we detect another failure: $y = B \wedge w = E$. We have the flexibility of modifying either the value of y or the value of w. Suppose we decide to modify w again. We then get a second eliminating explanation for w: $y = B \Rightarrow w \neq D$. Now, since both values of w have been eliminated, we have to change some other variable. In particular, the two eliminating explanations of w can be resolved, with the domain constraint of w, $w = D \vee w = E$ to derive a new failure explanation: $x = A \wedge y = B$. This essentially says that unless we change the value of x or y, we cannot find an assignment for w. (Notice that this is exactly what we get as the interior node explanation if we use the propagation procedure.) We now have the flexibility of modifying either the value of x or the value of y. If we want to modify the value of x (which we couldn't have done in normal search-tree based backtracking), all we need to do is to rewrite the failure explanation as $y = B \Rightarrow x \neq A$, and then change the value of x. Our new search node will look as follows:

$$
\begin{aligned}
N'' : \quad & \mathit{Assignment} : x = B, y = B, v = B, u = C, w = D \\
& \mathit{Eliminating\ Exp} : y = B \Rightarrow x \neq A; y = B \Rightarrow w \neq D; y = B \Rightarrow v \neq A; \\
& v = B \Rightarrow u \neq B; y = B \Rightarrow u \neq A
\end{aligned}
$$

Notice that we kept the values of y, v, u as well as w since they are not eliminated by the eliminating explanations of the current node. The eliminating explanation $x = A \Rightarrow w \neq E$ is removed since it is not relevant once x became B.

One caveat here is that since we only remember the nogoods relevant to the current partial assignment, if we are not careful, we can get into looping and fail to terminate. Specifically, when the irrelevant explanation $x = A \Rightarrow w \neq E$ is removed, there is a possibility that at a latter time, we may encounter another failure involving x and w (such as $x = B \wedge w = E$, in N'') and decide to change the value of x this time (by writing the failure as the eliminating explanation $w = E \Rightarrow x \neq B$), and consequently make w a past variable of x. This can lead to cycling behavior[28] since we had earlier made x a past variable of w.

The problem here is that when we removed $x = A \Rightarrow w \neq E$, we also forgot that we had made x past variable compared to w. A simple solution is to remember, whenever we rewrite an explanation of failure into an eliminating explanation for a variable x', that all the variables on the left hand side of the eliminating explanation are effectively made past variables of x'. This can be done by maintaining, in addition to the eliminating explanations, a partial order among variables, which is monotonically refined. Even when an eliminating explanation becomes irrelevant and leaves the node, the past-future variable distinctions it made become part of the partial order. When new failures are encountered, we must respect the partial order among the variables when converting them into eliminating explanations. This effectively avoids cycling and ensures termination.

Doing this in our example would make us accumulate the following partial orderings by the time we get to N'': $x \prec w$, $y \prec x$, $y \prec v$ and $y \prec v$. Thus the complete description of N'' is :

$$
\begin{aligned}
N'' : \quad & \mathit{Assignment} : x = B, y = A, v = B, u = C, w = D \\
& \mathit{Eliminating\ Exp} : y = B \Rightarrow x \neq A; y = B \Rightarrow w \neq D; \\
& y = B \Rightarrow v \neq A; v = B \Rightarrow u \neq B; y = B \Rightarrow u \neq A \\
& \mathit{Partial\ order} : (x \prec w, y \prec x, y \prec v, y \prec v).
\end{aligned}
$$

Since $x \prec w$ is part of all the descendants of N'', we will never be able to make w a past variable of x.

This is the idea behind partial order dynamic backtracking (Ginsberg & McAllester, 1994). The original dynamic backtracking algorithm due to Ginsberg (Ginsberg, 1993) was

28. To see this, note that at some future point, y's value may be changed, making $y = B$) x 6= A irrelevant, and at that time, nothing stops us from reverting back to $x = A$ and re-trace the failures between x and w.

less general than this– it assumed a pre-specified total order on the variables, as against an incrementally constructed partial order.

With the partial order among variables, the dynamic backtracking algorithm can be seen as effectively maintaining *several* search tree topologies simultaneously and backtracking on any one of them (depending on the heuristics), thus giving more flexibility in backtracking. Ginsberg and McAllester (Ginsberg & McAllester, 1994) argue that this flexibility provides the dynamic backtracking algorithm the ability to exploit local gradients in the search space like GSAT (Selman et al., 1992) and other local-search algorithms, without sacrificing completeness.

To complete the analogy with local search algorithms like GSAT, we note that since we don't need to erase the values of intermediate variables during backtracking, and since we can effectively backtrack to any variable, there really is no reason to search in the space of partial assignments. We can directly traverse the space of complete assignments. The decision as to which variables value should be changed can be made by independent heuristics such as "min-conflict" heuristic (Selman et al., 1992), that attempt to follow local gradients in the search space. Having decided which variable to change, we can then select a failure explanation involving that variable and rewrite the explanation such that it becomes an eliminating explanation for that variable. Of course, dynamic backtracking, being systematic, does not completely equal the freedom of movement provided by GSAT. Partial order dynamic backtracking provides unlimited freedom in selecting variables to reassign at the beginning of the search. However, as the search progresses, the partial order monotonically tightens, reducing the freedom (thus ultimately ensuring termination).

Despite the theoretical elegance of the idea, there is as yet no clear evidence indicating that dynamic backtracking leads to improvements over conventional dependency directed backtracking in practice. In fact, Bayardo and Schrag (Bayardo & Schrag, 1996) report that simple EDB, coupled with relevance-based learning, and forward checking outperform dynamic backtracking in the test suites that Ginsberg and McAllester (Ginsberg & McAllester, 1994) present in favor of dynamic backtracking. Another problem is that dynamic backtracking is not as readily applicable to planning as the simpler EDB is. The main issue, as we remarked earlier, is the complexity of eliminating explanations.

A.3 EBL in Dynamic Backtracking:

Since dynamic backtracking algorithms use nogoods as part of the node representation, it would seem that they may not benefit much from learning. In fact, we are not aware of any evaluations of dynamic backtracking complemented with learning. Part of the confusion comes from not recognizing the difference between eliminating explanations (also called directional nogoods in (Ginsberg & McAllester, 1994)), and the learned nogoods. As we saw in Section A.2, eliminating explanations are stored only to give information about the position of the current node in the search space. It is enough to remember a polynomial number of them in the current node if we are willing to do partial order dynamic backtracking. In addition to the eliminating explanations, dynamic backtracking can benefit from learned nogoods, which become part of the problem constraints, and will be used to detect failures in the current assignment. The utility of the learned nogoods is of course governed by the tradeoff between the cost of storage and matching on one hand and reduction in search through learning on the other. Although the same "failure explanation" can become a part of both the eliminating explanations and the stored nogoods, whether or not it stays in the former is determined by the relevance considerations imposed by dynamic backtracking algorithm, while whether or not it stays in the latter is best determined by the EBL storage tradeoffs (see Section 4.5).

Indeed, the failure explanations comprising the eliminating explanations are exactly the explanations that will be stored by a 0^{th}-order relevance based learning scheme (Bayardo & Miranker, 1996). Bayardo and Schrag's recent empirical results suggest that 4^{th}-order relevance based learning leads to better performance. This suggests that dynamic backtracking algorithms can benefit stored nogoods other than the directional ones included in the search node. Although Bayardo and Schrag (Bayardo & Schrag, 1996) show that normal search using forward checking, EDB and EBL outperforms dynamic backtracking, it would be interesting to see whether the dominance holds when Dynamic backtracking is armed with EBL.

Contents