Formalizing Dependency Directed Backtracking and Explanation Based Learning in Refinement Search

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

Abstract

The ideas of dependency directed backtracking (DDB) and explanation based learning (EBL) have developed independently in constraint satisfaction, planning and problem solving communities. 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. I show that both DDB and EBL depend upon the common theory of explaining search failures, and regressing them to higher levels of the search tree. The relevant issues of importance include (a) how the failures are explained and (b) how many failure explanations are remembered. This task-independent understanding of DDB and EBL helps support cross-fertilization of ideas among Constraint Satisfaction, Planning and Explanation-Based Learning communities.

1 Introduction

One of the main-stays of AI literature is the idea of "dependency directed backtracking" as an antidote for the inefficiencies of chronological backtracking [16]. However, there is a considerable confusion and variation regarding the various implementations of dependency directed backtracking. Complicating the picture further is the fact that many "speedup learning" algorithms that learn from failure (c.f. [10; 1; 9]), do analyses that are quite close to the type of analysis done in the dependency directed backtracking algorithms. It is no wonder then that despite the long acknowledged utility of DDB, even the more comprehensive AI textbooks such as [15] fail to provide a coherent account of dependency directed backtracking. Lack of a coherent framework has had ramifications on the research efforts on DDB and EBL. For example, the DDB and speedup learning techniques employed in planning and problem solving on one hand [10], and CSP on the other [3; 17], have hither-to been incomparable.

My motivation in this paper is to put the different ideas and approaches related to DDB and EBL in a common perspective, and thereby delineate the underlying commonalities between research efforts that have so far been seen as distinct. To this end, I consider all backtracking and learning algorithms within the context of general refinement search [7]. 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 DDB and EBL depend upon the 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. DDB occurs any time the explanation of failure regresses unchanged over a refinement decision. EBL involves remembering the interior node failure explanations and using them in the future. The relevant issues of importance include how the failures are explained, and how many of them are stored for future use. I will show how the existing methods for DDB and EBL vary along these dimensions. I believe that this unified task-independent understanding of DDB and EBL helps support cross-fertilization of ideas among the CSP, planning and EBL communities.

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 dependency directed backtracking and explanation based learning in refinement search. In Section 4, I discuss several variations of the basic DDB/EBL techniques. In Section 5, I relate this method to existing notions of dependency directed backtracking and explanation based learning in CSP and planning. Section 6 summarizes our conclusions.

2 Refinement Search Preliminaries

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 $\mathcal{N}$ 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) **R**, and a solution constructor function `sol`. The search process starts

*This research is supported in part by NSF research initiation award (RIA) IRI-9210997, NSF young investigator award (NYI) IRI-9457634 and ARPA/Rome Laboratory planning initiative grants F30602-93-C-0039 and F30602-95-C-0247. The ideas described here developed over the course of my interactions with Suresh Katukam, Gopi Bulusu and Yong Qu. 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.

Algorithm Refine-Node($\mathcal{N}$)
Parameters: (*i*) sol: Solution constructor function.
(*ii*) **R**: Refinement operators.
(*ii*) **CE**: fn. for computing the explanation of failure.

0. **Termination Check:**
If sol($\mathcal{N}$) returns a solution, return it, and terminate.
If it returns $*fail*$, fail.
Otherwise, select a flaw F in the node $\mathcal{N}$.
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
$\mathcal{N}' \leftarrow d_i(\mathcal{N})$
If $\mathcal{N}'$ is inconsistent
fail.
Compute **CE**($\mathcal{N}'$) *the explanation of failure for* $\mathcal{N}'$; **Propagate**($\mathcal{N}'$)
Else, **Refine-Node**($\mathcal{N}'$).

Figure 1: General template for Refinement search. The underlined portion provides DDB and EBL capabilities.

with the initial node $\mathcal{N}_\emptyset$, 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 $\{\mathcal{N}_1, \mathcal{N}_2, \cdots \mathcal{N}_n\}$, called refinements of $\mathcal{N}$, such that the candidate set of each of the refinements is a subset of the candidate set of $\mathcal{N}$. Each complete refinement operator can be thought of as corresponding to a set of decisions $d_1, d_2, \cdots, d_n$ such that $d_i(\mathcal{N}) = \mathcal{N}_i$. 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.

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 $\mathcal{N}$ 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 $\mathcal{N}$. The search process thus 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) [17] 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 $C_i(X_{i_1}, \cdots, X_{i_j})$ is a subset of the cartesian production $D_{i_1} \times \cdots \times D_{i_j}$ 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.

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 $\mathcal{N}$ (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 $d^i_1, d^i_2, \cdots, d^i_{|D_i|}$. Each decision d^i_j corresponds to *adding* the constraint $X_i = D_i[j]$, (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 is unassigned in A.
Effects: $\mathcal{A} \leftarrow \mathcal{A} + (x_i \leftarrow v_j^{x_i})$

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. The solution is any sequence of actions such that executing those actions from the initial state, in that sequence, will lead us to goal state.

Search nodes in planning can be represented (see [7]) as 6-tuples $\langle S, \mathcal{O}, \mathcal{B}, \mathcal{L}, \mathcal{E}, \mathcal{C} \rangle$, consisting of a set of steps, orderings, bindings, auxiliary constraints, step effects and step preconditions. These constraint sets, called partial plans, are shorthand notations for the set of ground operator sequences that are consistent with the constraints of the partial plan.

There are several types of complete refinement operators in planning [8], 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 C that needs to be true at a step s in a partial plan P, and making a set of children plans $P_1 \cdots P_n$ such that in each P_i, there exists a step s' that precedes s, which adds the condition C. P also contains, (optionally) a "causal link" constraint $s' \xrightarrow{C} s$ to protect C between s' and s. Each refinement P_i corresponds to an establishment decision d_i, such that d_i adds the requisite steps, orderings, bindings and causal link constraints to the parent plan to produce P_i. Once again, we can represent this decision as an operator with preconditions and effects.

3 Basic formulation of DDB and EBL

In this section, we will look at the formulation of DDB 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

Problem	Nodes	Candidate Set	Refinements	Flaws	Soln. Constructor
CSP	Partial assignment $\mathcal{A}$	Complete assignments consistent with $\mathcal{A}$	Assigning values to variables	Unassigned variables in $\mathcal{A}$	Checking if all variables are assigned in $\mathcal{A}$
Planning	Partial plan $\mathcal{P}$	Ground operator sequences consistent with $\mathcal{P}$	Establishment, Conflict resolution	Open conditions, Conflicts in $\mathcal{P}$	Checking if any ground linearization of $\mathcal{P}$ is a solution

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

Procedure Propagate($\mathcal{N}_i$)

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

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

Figure 3: The complete procedure for propagating failure explanations and doing dependency directed backtracking

does not learn from its failures, and can thus repeat the same failures in other branches. DDB 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. The procedure **Propagate** in Figure 3 shows how this is done. In the following we explain this procedure. Section 3.1 illustrates this procedure with an example from CSP.

Suppose a search node $\mathcal{N}$ 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 $\mathcal{N}'$ of $\mathcal{N}$ such that the decision taken under $\mathcal{N}'$ has had some consequence on the detected failure. To implement this approach, we need to sort out the relation between the failure at $\mathcal{N}$ and the refinement decisions leading to it. We can do this by declaratively characterizing the failure at $\mathcal{N}$.

Explaining Failures: From the refinement search point of view, a search node $\mathcal{N}$ 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 $\mathcal{N}$ is *empty* (because of an inconsistency among the constraints of $\mathcal{N}$), or because the constraints of $\mathcal{N}$ together with the global constraints of the problem, 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 either because the ordering and binding constraints comprising it are inconsistent by themselves, or violate the domain axioms. In either case, we can associate the failure at $\mathcal{N}$ with a subset of constraints in $\mathcal{N}$, say E, which, possibly together with some domain constraints δ, causes the inconsistency (i.e., $\delta \wedge E \models False$). E is called the explanation of failure of $\mathcal{N}$.

Suppose $\mathcal{N}$ is the search node at which backtracking was necessitated. Suppose further that the explanation for the failure at $\mathcal{N}$ is given by the set of constraints E (where E is a subset of the constraints in $\mathcal{N}$). Let $\mathcal{N}_p$ be the parent of search node $\mathcal{N}$ and let d be the search decision taken at $\mathcal{N}_p$ that lead to $\mathcal{N}$. We want to know whether d played any part in the failure of $\mathcal{N}$, and what part of $\mathcal{N}_p$ was responsible for the failure of $\mathcal{N}$ (remember that the constraints in $\mathcal{N}$ are subset of the constraints of its parent). We can answer these questions through the process of regression.

Regression: Formally, regression of a constraint c over a decision d is the set of constraints that must be present in the partial plan before the decision d, such that c is present after taking the decision.[1] Regression of this type is typically studied in planning in conjunction with backward application of STRIPS-type operators (with add, delete, and precondition lists), and is quite well-understood (see [12]). Here I adapt the same notion to refinement decisions as follows:

Regress(c, d)
= True if $c \in effects(d)$
= c' if $c'' \in effects(d)$ and $(c'' \wedge c') \vdash c$
= c Otherwise
Regress($c_1 \wedge c_2 \cdots \wedge c_n, d$)
= $Regress(c_1, d) \wedge Regress(c_2, d) \cdots \wedge Regress(c_n, d)$

Dependency Directed Backtracking: Returning to our earlier discussion, suppose the result of regressing the explanation of failure E of node $\mathcal{N}$, over the decision d leading to $\mathcal{N}$, $d^{-1}(E)$, be E'. Suppose $E' = E$. In such a case, we know that the decision d did not play any role in causing this failure.[2] Thus, there is no point in backtracking and trying another alternative at $\mathcal{N}_p$. This is because our reasoning shows that the

[1] Note that in regressing a constraint c over a decision d, 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 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.

[2] Equivalently, DDB can also be done by backtracking to the highest ancestor node of $\mathcal{N}$ which still contains all the constraints in E. I use the regression based model since it foregrounds the similarity between DDB and EBL.

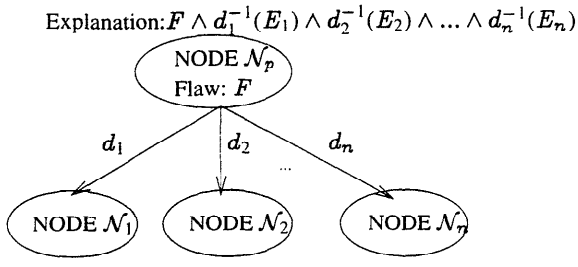

Figure 4: Computing Failure Explanations of Interior Nodes

constraints comprising the failure explanation E are present in $\mathcal{N}_p$ also, and since by definition E is a set of inconsistent constraints, $\mathcal{N}_p$ is also a failing node. This reasoning forms the basis of *dependency directed backtracking*. Specifically, in such cases, we can consider $\mathcal{N}_p$ as failing and continue backtracking upward using the **propagate** procedure, and using E as the failure explanation of $\mathcal{N}_p$.

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 $\mathcal{N}$ did have an effect on the failure in $\mathcal{N}$. At this point, we need to consider the sibling decisions of d under $\mathcal{N}_p$. If there are no unexplored sibling decisions, this again means that all the refinements of $\mathcal{N}_p$ have failed. The failure explanation for $\mathcal{N}_p$ can be computed in terms of the failure explanations of its children, and the flaw that was resolved from node $\mathcal{N}_p$, as shown in Figure 4.

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. The failure explanation thus computed can be used to continue propagation and backtracking further up the search tree. Of course, if any of the explanations of the children nodes of $\mathcal{N}_P$ regress unchanged over the corresponding decision, then the explanation of failure of $\mathcal{N}_P$ will be set by DDB as that child's failure explanation.

Explanation Based Learning: Until now, we talked about the idea of using failure explanations to assist in dependency directed 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 $\mathcal{N}_p$ 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 $\mathcal{N}'$ in another search branch such that E_p is a subset of $\mathcal{N}'$, then we can consider $\mathcal{N}'$ to be failing with E_p as its failure explanation. A variation of this approach involves learning search control rules [10] which recommend rejection of individual decisions of a refinement operator if they will lead to failure. When the child $\mathcal{N}_1$ of the search node $\mathcal{N}_p$ 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.[3]

[3]Explanation-based learning normally also involves a generalization step, where the failure explanation is generalized by replacing constants with variables [9]. Although such generalization can be very important in supporting inter-problem transfer, addition of generalization steps does not have a crucial impact on the analysis given in this paper. See [9] for a comprehensive discussion of the issues involved in explanation generalization.

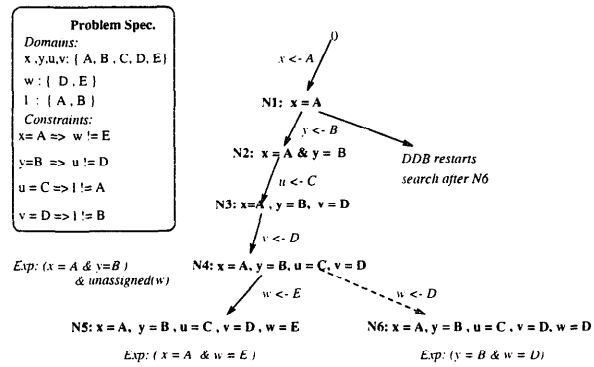

Figure 5: A CSP example to illustrate DDB and EBL

Unlike DDB, 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. 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 Problem in the Machine learning community [11; 6]. We shall review various approaches to it later.

3.1 Example

Let me illustrate the DDB and EBL process above with a simple CSP example shown in Figure 6 (for a planning example that follows the same formalism, see the discussion of UCPOP-EBL in [9]). 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 figure shows a series of refinements culminating in node N5, which is a failure node. An explanation of failure of N5 is $x = A \wedge w = E$ (since this winds up violating the first constraint). This explanation, when regressed over the decision $w \leftarrow E$ that leads to N5, becomes $x = A$ (since $w = E$ is the only constraint that is added by the decision). Since the explanation changed after regression, we restart search under N4, and generate N6. N6 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 N5, and the flaw description at N5 to give the explanation of failure of N4 as $E(N4) : x = A \wedge y = B \wedge \mathit{unassigned}(w)$. At this point $E(N4)$ can be remembered as a learned failure explanation (aka nogood [16]), and used to prune nodes in other parts of the search tree. Propagation progresses upwards. The decision $v \leftarrow D$ does not affect the explanation $N4$, and thus we backtrack over the node N3, without refining it further. Similarly, we also backtrack over N2. $E(N4)$ does change when regressed over $y \leftarrow B$ and thus we restart search under N1.

4 Variations on the DDB and EBL Theme

The basic approach to DDB and EBL that we described in the previous section admits several variations based on how

the explanations are represented, selected and remembered. I discuss these variations below.

4.1 Selecting a Failure Explanation

In our discussion of DDB 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 DDB, and the utility of the EBL rules learned. The most obvious explanation of failure of a dead-end node $\mathcal{N}$ is the set of constraints comprising $\mathcal{N}$ itself. In the example in Figure 5, $E(N5)$ 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 $\mathcal{N}$ as the explanation of its own failure makes DDB degenerate into chronological backtracking* (since the node $\mathcal{N}$ must have been affected by every decision that lead to it[4]). Furthermore, given the way the explanations of failure of the interior nodes are computed (see Figure 4), no ancestor $\mathcal{N}'$ of $\mathcal{N}$ can ever have an explanation of failure simpler than $\mathcal{N}'$ 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. For example, in CSP, a domain 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, ordering and binding inconsistencies can be used as starting failure explanations in planning.

Often, there may be multiple possible failures of explanation for a given node. For example, in the example in Figure 5, suppose we had another constraint saying that $u = C \Rightarrow W \neq E$. In such a case, the node N5 would have violated two different constraints, and would have had two failure explanations -- $E_1 : x = A \wedge W = E$ and $E_2 : u = C \wedge w = E$. In general, it is useful to prefer explanations that are smaller in size, or explanations that refer to constraints that have been introduced into the node by earlier refinements (since this will allow us to backtrack farther up the tree). By this argument E_1 above is preferable to E_2 since E_2 would have made us backtrack only to N2, while E_1 allows us to backtrack up to $\mathcal{N}_1$. These are however only heuristics. It is possible to come up with scenarios where picking the lower level explanation would have helped more.

4.2 Remembering (and using) Learned Failure Explanations

Another issue that is left open by our DDB/EBL algorithm is exactly how many learned failures should be stored. Although this decision does not affect the soundness and completeness of the search, it can affect the efficiency. 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. CSP, and machine learning literatures took differing approaches to this problem. Researchers in CSP (e.g. [3; 17]) concentrated on the syntactic characteristics of the nogoods, such as their size and minimality, to decide whether or not they should be stored. Researchers in machine learning concentrated instead on approaches that use the distribution of the encountered problems to dynamically modify the stored rules (e.g. by forgetting ineffective rules) [11; 6]. These differences are to some extent caused by the differences in CSP and planning problems. The nogoods learned in CSP problems have traditionally only been used in intra-problem learning, to cut down search in the other branches of the same problem. In contrast, work in machine learning concentrated more on inter-problem learning. (There is no reaon for this practice to continue however, and it is hoped that the comparative analysis here may in fact catalyze inter-problem learning efforts in CSP).

[4] we are assuming that none of the refinement decisions are degenerate; each of the add at least one new constraint to the node.

5 Relations to existing work

Figure 6 provides a rough conceptual flow chart of the existing approaches to DDB and EBL in the context of our formalization. In the following we will discuss differences between our formalization and some of the implemented approaches. Most CSP techniques do not explicitly talk about regression as a part of either the backtracking or learning. 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. For example, a constraint $x = a$ must have been added by the decision $x \leftarrow a$. Thus, in the example in Figure 5 it would have been easy enough to see that we can "jump back" to N1 as soon as we computed the failure explanation at N4. This sort of direct correspondence has facilitated specialized versions of DDB algorithm that use "constraint graphs" and other syntactic characterizations of a CSP problem to help in deciding which decision to backtrack to [17]. Regression is however important in other refinement search scenarios including planning where there is no one-to-one correspondence between decisions and the constraints in the node.

Most CSP systems do not add the flaw description to the interior node explanations. This makes sense given that most CSP systems use learned explanations only within the same problem, and the same flaws have to be resolved in every branch. The flaw description needs to be added to preserve soundness of the learned nogoods, if these were to be used across problems. The flaw description is also important in planning problems, even in the case of intra-problem learning. where different search branches may involve different subgoaling structures and thus different flaws.

Traditionally learning of nogoods in CSP is done by simply analyzing the dead-end node and enumerating all small subsets of the node assignment that are by themselves inconsistent. The resultant explanations may not correspond to any single explicit violated constraint, but may correspond to the violation of an entailed constraint. For example, in the example in Figure 5, it is possible to compute $u = C \wedge v = D$ as an explanation of failure of N5, since with those values in place, l cannot be given a value (even though l has not yet been considered until now). Dechter [3] shows that computing the minimal explanations does not necessarily pay off in terms of improved performance. The approach that we described in this paper allows us to start with any reasonable explanation of failure of the node -- e.g. a learned nogood or domain constraint that is violated by the node -- and learn similar minimal explanations through propagation. It seems plausible that the interior node failure explanations learned in this way are more likely to be applicable in other branches

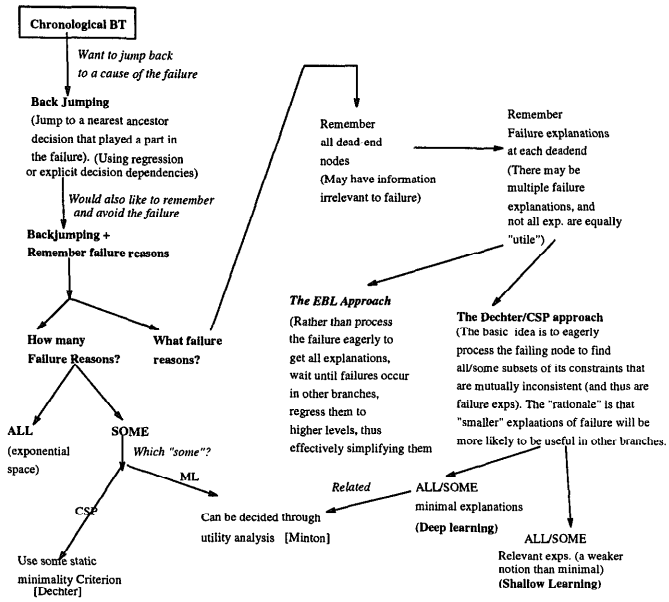

Figure 6: A schematic flow chart tracing the connections between implemented approaches to DDB and EBL

and problems since they resulted from the default behavior of the underlying search engine.

Intelligent backtracking techniques in planning include the "context" based backtracking search used in Wilkin's SIPE [18], and the decision graphs used by Daniels et. al. to support intelligent backtracking in Nonlin [2]. The decision graphs and contexts explicitly keep track of the dependencies between the constraints in the plan, and the decisions that were taken on the plan. These structures are then used to facilitate DDB. In a way, decision graphs attempt to solve the same problem that is solved by regression. However, the semantics of decision graphs are often problem dependent, and storing and maintaining them can be quite complex [14]. In contrast, the notion of regression and propagation is problem independent and explicates the dependencies between decisions on an as-needed basis. On the flip side, 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.

6 Summary

In this paper, we 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 DDB and EBL 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. DDB occurs when the explanation of failure regresses unchanged over a refinement decision, while EBL involves storing and applying failure explanations of the interior nodes in other branches of the search tree or other problems. I showed that the way in which the initial failure explanation is selected can have a significant impact on the extent and utility of DDB and EBL. The utility of EBL is also dependent on the strategies used to manage the stored failure explanations. I have also explained the relations between our formalization of DDB and EBL and the existing work in planning and CSP areas. It is hoped that this task-independent formalization of DDB/EBL approaches will clarify the deep connections between the two ideas, and also facilitate a greater cross-fertilization of approaches from the CSP, planning and problem solving communities. For example, CSP approaches could benefit from the results of research on utility of EBL, and planning research could benefit from the improved backtracking algorithms being developed for CSP [5].

References

[1] N. Bhatnagar and J. Mostow. *On-line Learning From Search Failures Machine Learning*, Vol. 15, pp. 69-117, 1994.

[2] L. Daniel. Planning: Modifying non-linear plans *University Of Edinburgh, DAI Working Paper: 24*

[3] R. Dechter. Enhancement schemes for learning: Back-jumping, learning and cutset decomposition. *Artificial Intelligence*, Vol. 41, pp. 273-312, 1990.

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

[5] M. Ginsberg and D. McAllester. GSAT and Dynamic Backtracking. In *Proc. KRR*, 1994.

[6] J .Gratch and G. DeJong COMPOSER: A Probabilistic Solution to the Utility problem in Speed-up Learning. In *Proc. AAAI 92*, pp:235--240, 1992

[7] S. Kambhampati, C. Knoblock and Q. Yang. Planning as Refinement Search: A Unified framework for evaluating design tradeoffs in partial order planning. *Artificial Intelligence* special issue on Planning and Scheduling. Vol. 76, pp/ 167-238. 1995.

[8] S. Kambhampati and B. Srivastava. Universal Classical Planner: An Algorithm for unifying state-space and plan-space planning. In *Proc. 3rd European Workshop on Planning Systems*, 1995.

[9] S. Kambhampati, S. Katukam and Y. Qu. Enclosed please find three copies of our paper "Failure driven dynamic search control for partial order planners: An explanation-based approach" *Artificial Intelligence*, Fall 1996. (To appear).

[10] S. Minton, J.G Carbonell, C.A. Knoblock, D.R. Kuokka, O. Etzioni and Y. Gil. Explanation-Based Learning: A Problem Solving Perspective. *Artificial Intelligence*, 40:63--118, 1989.

[11] S. Minton. Quantitative Results Concerning the Utility of Explanation Based Learning. *Artificial Intelligence*, 42:363--391, 1990.

[12] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, 1980.

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

[14] C. Petrie. Constrained Decision Revision. In *Proc. 10th AAAI*, 1992.

[15] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[16] R. Stallman and G. Sussman. Forward Reasoning and Dependency-directed Backtracking in a System for Computer Aided Circuit Analysis. *Artificial Intelligence*, Vol. 9, pp. 135-196. 1977.

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

[18] D. Wilkins. *Practical Planning.* Morgan Kaufmann (1988).