Failure-Driven Refinement Search with Local Repair-Based Heuristics for Constraint Satisfaction Problems

Chia-Lin Hsieh

(Received November 15, 1999; Revised January 6, 2000; Accepted January 11, 2000)

Abstract

There has been substantial recent interest in two new families of search techniques. One family consists of systematic approaches, which use the idea of dependency directed backtracking (DDB) and dynamic backtracking (DB) as an antidote for the inefficiencies of chronological backtracking. Based on backtracking, systematic approaches are able to produce an optimal assignment, when no time limit is imposed. The other contains non-systematic methods such as Min-Conflict Repair Heuristic or Selman’s GSAT. Based on local improvement mechanisms, these methods can’t guarantee optimality, but may produce better quality assignments in a limited time. The main attractions are their reactivity and applicability to optimization problems. Our aim in this paper is to describe a new search procedure that combines the benefits of both of the earlier approaches.

To achieve this goal, we describe a complete hybrid method called failure driven search control with min-conflict repair (FDB-MC) for solving constraint satisfaction problems. The failure explanation-backmarking process controls the search space of this combined approach, a method of learning constraints during search (at each failure point) that may be used to avoid the repeated traversing of failed path in a search tree. The approach is based on repair-based algorithms, and can be proved to find an optimal solution.

Keywords: constraint satisfaction problem (CSP), failure-driven backtracking (FDB), explanation-based learning (EBL), min-conflict heuristic, and repair-based search

1. Introduction

Constraint satisfaction involves finding values for problem variables subject to constraints on acceptable combinations of values. The constraint satisfaction problem (CSP) model is widely used to represent and solve various AI related problems such as scheduling, planning, map coloring or computer aided design and provides fundamental tools in areas such as truth maintenance, expert systems or constraint logic programming. These problems are difficult because they involve search; there is never a
guarantee that (for example) a successful coloring of a portion of a large map can be extended to a coloring of the map entirely.

According to Ginsberg and McAllester’s paper [1], the algorithms developed recently have been of two types. Systematic algorithms decide whether a solution exists by searching the entire space. In recent years, there has been great emphasis on developing methods for improving backtracking efficiency. The term “failure-driven backtracking” (FDB) or “dependency-directed backtracking” (DDB) [2] denote such improvements. Unfortunately, despite the long acknowledged utility of DDB, it fails to provide a coherent account of DDB, especially a lack of a coherent framework on the efforts on DDB and learning methods (e.g. Explanation-based Learning (EBL) [3] or Justification Based Learning (JBL) [4]).

The other type of algorithms, called Local algorithms, use hill-climbing techniques to find a solution quickly but are non-systematic in that they search the whole space based on a probabilistic sense. These non-systematic algorithms appear to be empirically effective because of their ability to follow the local gradients based on the number of violated constraints in the search space.

Dynamic backtracking [5] tries to overcome these problems by retaining specific information about the portions of the search space that have been eliminated and then following local gradients in the remainder. Unlike previous algorithms that recorded such elimination information such as dependency-directed backtracking [6] and backjumping [7], dynamic backtracking is selective about the information so that only a polynomial number of memories are required. Unfortunately, neither dynamic nor dependency-directed backtracking (or any other similar methods) is truly effective at local gradients within the search space, since the basic underlying methodology remains simply chronological backtracking. New techniques are developed to make the search more efficient, but the complexity of the search is still exponential.

The second group of algorithms developed recently presumes that the degree of movement is greater than systemacity. In order to achieve the freedom of movement, the algorithms consider the search space as a space of total assignments of values to variables. Changes are allowed between any two assignments that differ on a single value, and a hill-climbing procedure is employed to try to minimize the number of constraints violated by the overall assignment. The best known algorithms of this type are Min-Conflict [8] and GSAT [9].
Min-Conflicts has been applied mainly to the scheduling domain. These repair-based techniques usually work on complete but may cause inconsistency. It is naturally extensible to reactive scheduling and optimization since it always repairs on a complete assignment. GSAT is limited to prepositional satisfiability problems (where every variable is assigned simply true or false), and has led to remarkable progress in the solution of randomly generated problems of this type. A key issue here is to avoid the trapping of a local minimum in a repair process. It has been found that these repair techniques are often suited to problems with a large solution space with a high ratio of inconsistency.

These techniques have been proved empirically and theoretically successful for many applications. However, it is a commonly acknowledged fact that no single technique is universally good for all constraint problems. Our aim in this paper is to describe a new search procedure that appears to combine the benefits of both of the earlier approaches. To achieve this goal, we describe a complete hybrid method called failure driven search control with min-conflict repair (FDB-MC) for solving constraint satisfaction and optimization problems.

The next section describes some preliminary definitions of the CSP models, and the usual techniques applied to their solution. Section 3 gives a presentation of the original Dynamic Backtracking algorithm [5]. Section 4 reviews a unified approach of refinement search to show how CSP can be modeled in terms of refinement search. Furthermore, a method for formulating FDB and EBL in refinement search is provided. Section 5 extends the basic FDB framework to give more flexibility during backtracking. Based on this framework, a novel algorithm called (FDB-MC) is proposed. Section 6 compares our proposed methodology with other well-known techniques and suggests some possible modifications that might enhance FDB-MC. Section 7 summarizes the contributions of the paper, and discusses how the potential of the improved algorithm can be used in some of the application domains.

2. Definitions and Preliminaries

2.1 Constraint Satisfaction Problems

**Definition 1.** A constraint satisfaction problem (CSP) \((V, C, R_c)\) involves a set \(V=\{V_1, V_2, \ldots, V_n\}\) of \(n\) variables and a set \(C\) of constraints. Each variable \(V_i \in V\) takes its value in its finite domain \(D_i\). Each constraint \(C_p\) involves a subset \(V_{cp} = \{V_{1p}, V_{2p}, \ldots, V_{lp}\}\) of
$V$ and is defined by a relation $R_c$ of the Cartesian product $D_{1p} \times D_{2p} \ldots \times D_{qp}$ that specifies which values of the variables are compatible with each other.

A CSP may be associated with its constraint hyper-graph in which nodes represent variables and hyper-arcs represent constraints. A binary CSP is a CSP whose constraints are binary (i.e., involve at most two variables). A CSP can be associated with a constraint-graph in which nodes represent variables and arcs connect those pair of variables for which constraints are given. Let us consider the CSP represented in Figure 1 (modified from [11]). Each node represents a variable whose domain is explicitly indicated, and each arc is labeled with the set of value-pairs permitted by the constraints.

**Definition 2.** An assignment $\Gamma$ of values to variables in $Y \subset V$ satisfies a constraint $C_p$ such that $V_{Cp} \subset Y$ iff the set of value-pairs taken by the variables of $V_{Cp}$ in $\Gamma$ belongs to its associated relation $R_c$. An assignment of values to a subset $Y$ of the variables is consistent iff it satisfies all the constraints $C_p$ such that $V_{Cp} \subset Y$. A solution of a CSP is an assignment of values to all the variables such that all the constraints are satisfied.

The usual work in a CSP is to find one solution or all solutions. The problem is NP-hard. It may involve a backtracking algorithm that assigns consistent values to a subset of variables and try to extend to it a new instantiation such that the whole set is consistent.

The variable ordering, which decides the order in which the variables to get are instantiated, usually has an important effect on the efficiency of the algorithm since each ordering defines how soon a constraint may be checked [10]. One way to improve backtracking efficiency is to make it possible for the algorithm to rapidly detect the assignments that will not lead to a solution. This may be done by adding induced constraints to the CSP either before or during search. Thus, adding an induced constraint to a CSP does not change its solution set and may enable some global inconsistency to be discovered as early as possible. This is usually called constraint
Failure-Driven Refinement Search with Local Repair-Based Heuristics for Constraint Satisfaction Problems

*propagation* and has been formalized in various levels of so-called *local consistency*: arc-consistency, path-consistency, and *k*-consistency [11].

### 2.2 The Dynamic Constraint Satisfaction Problem

Many AI synthesis problems such as planning, scheduling or design can be modeled as and encoded in constraint satisfaction problems (CSP). However, many of them are actually *dynamic*. That is, the set of constraints to consider evolves because of the environment, the user or other agents in the framework of a distributed system. The notion of dynamic CSP (DCSP) [12] has been introduced to represent them. Since all the possible changes to a CSP can be expressed in terms of constraint additions or removals, a DCSP is just a sequence of CSPs, where each one differs from the previous one by some constraints either added or removed.

#### 2.2.1 Dynamic Problems

To solve such DCSPs, it is always possible to solve each one from scratch, as it has been done for the first one. But this method, which remembers nothing from the previous reasoning, has some drawbacks: *inefficiency* and *instability*. Therefore, one needs methods which, starting from the previous solution and the previous reasoning, allow a new solution to be rapidly found, as close as possible to the previous one.

#### 2.2.2 Existing Approaches

The DCSP is currently the subject of numerous studies. We can distinguish three approaches:

- **Heuristic methods**, which consist of using any previous consistent assignment as a heuristic in the framework of the current CSP. To maintain a given level of local consistency, those authors [9][13] use a JTMS-like (Justified Truth Maintenance System) approach;

- **Local repair methods**, which consist of starting from any previous consistent assignment and of repairing it, using a sequence of local *modifications* [14][15];

- **Constraint Recording methods**, which consist of recording any kind of justified constraints to be deduced and memorized so that they can be reused in the framework of any new CSP which included this justification [16][17].
3. Dynamic Backtracking

Ginsberg [5] proposed a new algorithm, called dynamic backtracking, for solving CSPs. This algorithm can be viewed as an improvement of conflict-directed backjumping [2]. Conflict Directed Backjumping records, at each step of the search and for each variable $v$, the set of previously assigned variables whose assignments have reduced the domain of $V$. Such sets are called conflict sets and the space which is required to record them is $O(n^2)$, where $n$ is the number of variables. Let $v$ be a variable whose current domain is empty, let $E(v)$ be its conflict set and let $v'$ be the last variable in $E(v)$ avoiding to the assignment order. In such a situation, the algorithm does not backtrack to the variable which precedes $v$, but backjumps to $v'$ and all the variables between $v'$ and $v$ are unassigned. Empirical results have shown that this simple method often avoids much useless searching, especially for the under-constrained CSPs.

Dynamic Backtracking records the same kind of information, but a finer level. In spite of its name, dynamic backtracking does not deal with dynamic CSPs. The term dynamic here means that its backtracking mechanism allows the variables to be unassigned in an order which is different from the one which has been used to assign them. Dynamic backtracking uses a set of failure explanations to both recording information about the portion of the search space that has been eliminated and to record the current partial assignment being considered by the procedure. Termination, correctness and completeness of this algorithm have been proven [5]. But the backtracking mechanism included in Dynamic Backtracking offers other opportunities, particularly in the framework of Dynamic Constraint Satisfaction Problem (DCSP). This leads us to extend the notion of failure explanations in order to take into account constraints and variable domains as assumptions, as previously done with variable assignments. An extended dynamic backtracking mechanism based on the concept of “failure explanation” is proposed in the following section to show how it is used to produce inconsistency explanations and to deal with dynamic problems.

3.1 Constraints and Failure Explanations: Preliminaries

In order to understand the approach, we begin with some preliminary properties and examples about the failure explanations in CSPs.
Definition 4. (Definition of Failure Explanation) Let $\Gamma$ be an assignment of a subset $V$ of the CSP variables; $\Gamma$ is a failure explanation iff there is no CSP solution which contains $\Gamma$.

Property 1. Let $\Gamma = \{v_1 \leftarrow val_1, \ldots, v_k \leftarrow val_k\}$ be an assignment of a subset $V$ of the CSP variables, let $v$ be any variable in $V$ and $val$ be its value in $\Gamma$; $\Gamma$ is an failure explanation iff $\Gamma_{\backslash \{v\}} \rightarrow v \neq val$ or $\Gamma$ is an expression of the form

$$(v_1 \leftarrow val_1) \land \ldots \land (v_k \leftarrow val_k) \rightarrow v \neq val$$  \hspace{1cm} (1)

Or logically equivalent to the implicit constraint

$$-[(v_1 = val_1) \land \ldots \land (v_k = val_k) \land (v = val)]$$

One special failure explanation is the empty set, which is tautologically false. If this failure explanation can be derived from the given set of constraints, it follows that no solution exists for the problem. The typical way in which new failure explanations are obtained is by resolving together old ones. As an example, suppose that we have the following:

$$(v_1 \leftarrow a) \land (v_2 \leftarrow b) \rightarrow v \neq val_1$$
$$(v_1 \leftarrow a) \land (v_3 \leftarrow c) \rightarrow v \neq val_2$$
$$(v_2 \leftarrow b) \rightarrow v \neq val_3$$

where $val_1$, $val_2$, $val_3$ are the only values in the domain of $v$. It follows that we can combine these failure explanations to conclude that there is no solution with

$$(v_1 \leftarrow a) \land (v_2 \leftarrow b) \land (v_3 \leftarrow c)$$  \hspace{1cm} (2)

Moving $v$ to the conclusion of (2) gives us

$$(v_1 \leftarrow a) \land (v_2 \leftarrow b) \rightarrow (v_3 \neq c)$$

In general, we may conclude that $\Lambda_j(v_j \leftarrow val_j)$ is inconsistent iff for any specific $v_k$,

$$\Lambda_{j \neq k} (v_j \leftarrow val_j) \rightarrow v_k \neq val_k$$  \hspace{1cm} (3)

3.2 An Example

To see how this is done, consider the small map coloring problem depicted in Figure 2, from [5]. The map consists of five states in the United States: Alabama (AL), Colorado (CO), Florida (FL), Indiana (IN) and Oregon(OR). We assume fictitiously that the states border each other as shown in the figure, where states are noted as nodes and border one another if and only there is an arc connecting them.
In coloring the map, we can use the three colors: red, green and blue. We will typically the colors and states names to single letters in the obvious way. The following table gives a trace of how a typical dependency-directed backtracking scheme might be used in this problem. The elimination column will be discussed shortly.

<table>
<thead>
<tr>
<th>Alabama (AL)</th>
<th>Colorado (CO)</th>
<th>Florida (FL)</th>
<th>Indiana (IN)</th>
<th>Oregon (OR)</th>
<th>Add Failure Explanations</th>
<th>Elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>g</td>
<td>r</td>
<td>r</td>
<td></td>
<td>A ← r → F ≠ r</td>
<td>1</td>
</tr>
<tr>
<td>r</td>
<td>g</td>
<td>b</td>
<td>r</td>
<td></td>
<td>A ← r → I ≠ r</td>
<td>2</td>
</tr>
<tr>
<td>r</td>
<td>g</td>
<td>b</td>
<td>g</td>
<td></td>
<td>C ← g → I ≠ g</td>
<td>3</td>
</tr>
<tr>
<td>r</td>
<td>g</td>
<td>b</td>
<td>b</td>
<td>r</td>
<td>A ← r → O ≠ r</td>
<td>4</td>
</tr>
<tr>
<td>r</td>
<td>g</td>
<td>b</td>
<td>b</td>
<td>g</td>
<td>C ← g → O ≠ g</td>
<td>5</td>
</tr>
<tr>
<td>r</td>
<td>g</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>I ← b → O ≠ b</td>
<td>6</td>
</tr>
<tr>
<td>r</td>
<td>g</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>(A ← r) ∧ (C ← g) → I ≠ b</td>
<td>7, 6</td>
</tr>
<tr>
<td>r</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td>A ← r → C ≠ g</td>
<td>8, 3, 5, 7</td>
</tr>
</tbody>
</table>

We begin by coloring Alabama red and Colorado green then try to color Florida red as well. Since this violates the constraint that Alabama and Florida be different colors, no-goods (1) in the above is produced. Then we change Florida color to blue and turn to Indiana. Since Indiana can not be colored red or green, failure explanation (2) and (3) appear; the only remaining color for Indiana is blue.

Unfortunately, having colored Indiana blue, we cannot color Oregon. Three no-goods are generated in (4), (5) and (6), and we can resolve this because the three conclusions eliminate all of the possible colors for Oregon. The result is that there is no solution with \((A = r) \land (C = g) \land (I = b)\), which we rewrite as (7) above. This can in turn be resolved with (2) and (3) to get (8). The analysis can continue at this point to gradually determine that Colorado is red, Indiana could be green or blue, and Oregon must then be the color not chosen from Indiana. However, the problem with this approach is that the set of failure explanations grows monotonically, with a new failure...
explanation being added at every step. The number of the failure explanations stored grows linearly with the run time and presumably exponential with the size of the problem.

So it is quite important to delete failure explanations when they become irrelevant in the sense that their antecedents no longer match the partial solution in the question. In the example above, failure explanations can be eliminated as indicated in the final column of the trace. When we derive (7), we remove (6) since Indiana is no longer colored blue. When we derive (8), we remove all of the failure explanations with C = g in their antecedents. Thus the only information we retain is that Alabama red color precludes red for Florida, Indiana and Oregon (1,2 and 4) and also green for Colorado (8).

4. Formalization of Failure-Driven Backtracking and Explanation Based Learning in Refinement Search

Although the idea of "dependency directed backtracking" can be an antidote for the inefficiencies of chronological backtracking [6], however, there are still various implementations of explanation driven backtracking. Many different ideas, such backjumping, failure explanation learning and dynamic backtracking can all be concerned as "failure driven backtracking". It seems that many "quality learning" algorithms that learn from failure (c.f. [3][9][17][18][19][20]), do analyses the similar way as the type of analysis performed in explanation driven backtracking approaches.

In order to gain the benefits from the different ideas and approaches related to failure driven backtracking (FDB) and explanation-based learning (EBL), we consider all backtracking and learning algorithms within the context of refinement search [21]. Within the refinement search approach, both FDB and EBL depend on a common theory of explaining search failures, and regressing them to the higher levels of the search tree to compute explanations of failures of the interior nodes. It is reasonable to say that explanation driven backtracking can be considered as a specified form of failure driven backtracking (FDB). Most of the existing backtracking algorithms are a specialization or extension of this idea.

Within this framework, the important issues here are possible depending on how the failure can be represented, contextualized, and how many of them should be stored for the future use. We will discuss them in the following sections. In addition, we will
show how the idea of dynamic backtracking [1] is related to the ideas of FDB and EBL in order to obtain a unified model to solve CSP problems.

4.1. Refinement Search: Preliminaries

The refinement search 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 regarded 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 found from one of the set in limited time. Each candidate node $N$ in the refinement search thus corresponds to a set of candidates. The candidate set of the node is implicitly defined as the set of candidates that satisfy the constraints on the node.

![FIGURE 3. SCHEMATIC ILLUSTRATION OF REFINEMENT PROCESS](image)

Figure 3. shows a schematic diagram illustrating the refinement process. A refinement search consists of a set of refinement operators (or strategies) $R$ and a set of solution state functions $s$. The search process starts with the initial node $N_0$ which corresponds to the set of all candidates. The search process involves splitting and narrowing the set of all potential solutions until an acceptable solution for the problem is reached. A refinement operator $R$ chooses a search node $N$, and returns a set of search results $\{N_1,N_2,\ldots,N_n\}$, called refinement of $N$, such that the candidate set of each operator of the refinement is a subset of the candidate set of $N$. Each refinement operator can be considered as corresponding to a set of decisions $d_1,d_2,\ldots,d_n$ such that $d_i(N) = N_i$. Adding additional constraints to the current search node can derive each of decision.

To make a more efficient refinement search, we use the notion of "flaws" in search node. 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
the flaw by adding the missing constraints. Figure 4 provides a generalized template for refinement search sets.

\[
\text{Parameters}
\]
\[
\begin{align*}
\text{s} & : \text{Solution state function.} \\
\text{R} & : \text{Refinement Operators.} \\
\text{Fail-Exp} & : \text{Function for computing failure explanations}
\end{align*}
\]

\[
\text{Algorithm}
\]
\[
\text{s} \leftarrow \text{initial\_state}
\]
\[
\text{Procedure \textit{Refinement\_Node}(N)}
\]
\[
\text{current\_s} = s(N)
\]
\[
\text{if current\_s}(N) \text{ return a solution then terminate}
\]
\[
\text{else if it returns "fail" then fail}
\]
\[
\text{else select a flaw F in the Node N(current\_s)}
\]
\[
(\text{iterative refinement})
\]
\[
\text{Pick a refinement operator R \in R that can solve F}
\]
\[
\text{let R correspond to the n refinement decisions d1,d2,\ldots,dn}
\]
\[
\text{for } d_i = d_1,d_2,\ldots,dn \text{ do}
\]
\[
N' \leftarrow d_i(N)
\]
\[
\text{If N' is inconsistent, then}
\]
\[
\text{failure explanation fail}
\]
\[
\text{Compute Fail-Exp(N') ; the failure explanation for N'}
\]
\[
\text{Propagate(N')}
\]
\[
\text{else}
\]
\[
\text{Refinement\_Node(N')}
\]

FIGURE 4. GENERAL TEMPLATE FOR REFINEMENT SEARCH

In this template, the function "\text{Propagate(N')}" plays an important part of refinement splitting, for it propagates the constraint by narrowing the candidate set of the node without splitting it. Constraint propagation essentially derives these constraints and adds them to the node description. In the next section, we will show how CSP problems can be modeled in terms of refinement search.

4.1.1 Constraint Satisfaction as Refinement Search

CSP problems can be generalized and modeled in terms of refinement search. Each search node in CSP contains constraints of the form Vi = Di, which provides 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.

Each unassigned variable in the current partial assignment can be seen as a "flaw" to be resolved. There is a refinement operator RVi corresponding to each variable Vi, which generates refinements of a node N by assigning a value from Di to Vi. Thus, RVi
corresponds to an "OR" branch in the search space corresponding to decisions \(d_1, d_2, \ldots, d_{|D_i|}\). Each decision \(d_j^i\) corresponds to adding the constraint \(V_i = \text{val}_{i[j]}\) to the current partial assignment (where \(\text{val}_{i[j]}\) is the \(j\)th value in the domain of the variable \(V_i\)). We can encode this as an operator with preconditions and effects as follows:

\[
\text{ASSIGN}(\Gamma; V_i, \text{val}_{i[j]})
\]

\[
\begin{align*}
\Gamma &: \text{Partial assignment} \\
\text{Preconditions} &: V_i \text{ needs assignment in } \Gamma \\
\text{Effects} &: \Gamma \leftarrow \Gamma + (V_i \leftarrow \text{val}_{i[j]})
\end{align*}
\]

Constraint propagation involves deriving consequences of the node assignment constraints, given the background of problem constraints. Figure 5 illustrates the refinement search process in the map coloring (CSP) problem considered in Section 3.2.

4.1.2 Dynamic Constraint Satisfaction as Refinement Search

Dynamic CSPs are the generalization of CSP problems. 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:

\[
V_i = \text{val}_i \land V_j = \text{val}_j \land V_k = \text{val}_k \land \ldots \land V_m = \text{val}_m \Rightarrow \text{NEED_ASSIGNMENT}(V_n)
\]

This constraint states that if \(V_i, V_j, \ldots, V_m\) have the listed assigned values, then the variable \(V_n\) will need an assignment.
Stating that certain subsets of variables require assignments specifies the initial problem. These are called the active variables. 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 constraints active, adding "NEED_ASSIGNMENT" new flaws corresponding to those variables to the current node. In other word, new flaws may result from a refinement decision. Assignment decisions will thus have the following generic precondition/effect structure:

\[
\text{ASSIGN} (\Gamma, V_i, \text{val}_i[j])
\]

- **Preconditions:** $V_i$ needs assignment in $\Gamma$
- **Effects:** $\Gamma \leftarrow \Gamma + (V_i \leftarrow \text{val}_i[j])$

If $V_i = \text{val}_i$, $V_j = \text{val}_j$, ..., $V_m = \text{val}_m$

then $\text{NEED_ASSIGNMENT}(V_n)$

### 4.2 The Basic Formulation of FDB and EBL

In this section, we will look at the formulation of FDB and EBL in refinement search. The refinement search template provided in Figure 4 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 of the search tree. FDB can be seen as a solution to the first problem, while EBL is seen as the solution to the second.

### 4.2.1 Propagation Procedure for Refinement Search

In order to incorporate FDB and EBL within the refinement search template, we have to compute the failure explanations at dead-end nodes, and pass information over to the propagation procedure that can compute failure explanations of interior nodes efficiently at the leaf nodes. The flow chart of the propagation procedure is shown in Figure 6. The procedure Propagate is described in more detail in Figure 7.

From the refinement search point of view, a search node $N$ is said to be failing if its candidate set might not contain any solution. To explain failures, we can associate the failure at node $N$ with the assignments in $N$, say $\Gamma$, which together with the domain constraints, $C$, lead to an inconsistency (i.e., $\Gamma \wedge C \Rightarrow \text{False}$). The problem and domain-
independent characterization of this failure could be $\Gamma \land C$. For the map coloring problem shown in Figure 5, the search fails first at leaf node $N_3$, because of the violation between the assignment $\Gamma (A=\text{red} \land F = \text{red})$ and the constraint $C (A=\text{red} \Rightarrow F\neq\text{red})$ (or

$$[A=\text{red} \land F = \text{red} \land (A=\text{red} \Rightarrow F\neq\text{red})] \Rightarrow False).$$

So the full failure explanation will be $A=\text{red} \land F = \text{red} \land (A=\text{red} \Rightarrow F\neq\text{red})$.

### 4.2.2 Regression Procedure for Refinement Search

Given a failure node, in order to support explanation driven backtracking, we need to evaluate the effect of various decisions leading to the failure node. Further, having picked up an ancestor node to backtrack, it is interesting to find out which constraints of the node are irrelevant to the observed failure. Since failures are characterized in terms of failure explanations, regression of failure explanations over decisions will help to check which constraints in the failure explanations should be used.
Failure-Driven Refinement Search with Local Repair-Based Heuristics for Constraint Satisfaction Problems

Formally, regression of a constraint \( C \) over a decision \( d(N) \) is the set of constraints that must appear in the partial plan before the decision \( d \), such that \( C \) is present after taking the decision\(^2\). Regression of this type is typically found in planning in conjunction with the backward application of STRIPS-type operators. Here we adapt the same notion to refinement decisions as follows.

\[
\text{Regress}(C, d) = \begin{cases} 
\text{true} & \text{if } C \in \text{effects}(d) \\
C' & \text{if } C'' \in \text{effects}(d) \text{ and } (C'' \wedge C') \Rightarrow C \\
C & \text{otherwise}
\end{cases}
\]

\[
\text{Regress}(C_1 \wedge C_2 \wedge \ldots \wedge C_n, d) = \text{Regress}(C_1, d) \wedge \text{Regress}(C_2, d) \wedge \ldots \wedge \text{Regress}(C_n, d)
\]

In other words, regressing a constraint \( C \) over a decision \( d(N) \) results in true if the constraint \( C \) is added by \( d(N) \), and false if it is deleted by \( d \). This results in the set of constraint \( C' \) if \( C \) can be inferred from the effects of \( d \) and the set of constraints \( C'' \) that are present in \( N \). So the regression of a failure explanation is accomplished by regressing individual constraints comprising the failure explanation and conjoining the result.

4.2.3 A CSP Example to Illustrate FDB and EBL

Now, a simple CSP example is used to illustrate the FDB and EBL process shown in Figure 8. The problem contains 5 variables: \( X_1, X_2, X_3, X_4, X_5 \). The domains of the variables and constraints are shown in Figure 8. Figure 9 shows the refinement process terminating in node \( N_5 \), which is a failure node. The failure explanation of \( N_5 \) is

\[^2\text{Note that in regressing a constraint } C \text{ over a decision } d(N), \text{ we are interested in the weakest constraints that need to be true before the decision, so that } C \text{ will be true after the decision is taken.}\]
\{X1=a \land X5=e\}\} (since this violates the first constraint). This explanation, after the regression over the decision \(X5 \leftarrow e\) that leads to \(N5\), becomes \(X1=a\). Since the explanation changed after regression, we start to search under node \(N4\), and generate \(N6\). Unfortunately, \(N6\) is also a failing node and its failure explanation is \(\{X2=b \land X5=d\}\). When this explanation is regressed over the corresponding decision, we get \(X2=b\). This is then conjoined with the regressed explanation from \(N5\), and the flaw description at \(N5\) to give the failure of explanation of \(N4\) as \(E(N4):\{X1=a \land X2=b \land \text{Unassigned}(X5)\}\). At this point, \(E(N4)\) is remembered as a "learned failure explanation", and can be used to prune nodes in other parts of the search tree. Propagation progresses upwards. Since the decision \(X4 \leftarrow d\) does affect the explanation \(N4\), thus we can backtrack over the node \(N3\), without refining it further. Similarly, we also backtrack over \(N2\). \(E(N4)\) does change when we regress over \(X2 \leftarrow b\) and thus we restart search under \(N1\).

### 4.3 Variations on the Basic FDB and EBL Scheme

The basic approach to FDB and EBL admits many variations based on how the explanations are represented, selected and remembered. These variations are discussed below.

#### 4.3.1 Selecting Failure Explanations

It happens quite often that there are multiple explanations of failure for a dead-end node, and the selected explanation may have an impact on the extent of FDB and EBL. The most obvious failure explanation of a dead-end node \(N\) is the set of constraints consisting of \(N\) itself. It is not hard to see that using \(N\) as its own failure explanation makes a FDB degenerate into chronological backtracking. 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 CSPs, when a domain constraint is violated by a part of the current assignment, then that part of the assignment can be taken as a failure explanation.

Often, there are multiple possible failure explanations for a given node. For example, consider the modified example in Figure 8, shown in Figure 9, by adding a new constraint saying that \((X3 =c \Rightarrow X5 \neq e)\). In such cases, the node \(N5\) would violate two different constraints, and would have two failure explanations: \(E_5^1:\{X1=a \land X5=e\}\) and \(E_5^2:\{X3=c \land X5=e\}\). In general, it is useful to select explanations that are smaller in
Failure-Driven Refinement Search with Local Repair-Based Heuristics for Constraint Satisfaction Problems

size (for smaller explanations are likely to be applicable and useful in other situations) 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 $E_5^1$ above is preferable to $E_5^2$ since $E_5^2$ would make us backtrack only $N_2$, $N_1 : \{X_1 = a\}$

$N_2 : \{X_1 = a \land X_2 = b\}$

$N_3 : \{X_1 = a \land X_2 = b \land X_4 = d\}$

$N_4 : \{X_1 = a \land X_2 = b \land X_4 = d \land X_3 = c\}$

$N_5 : \{X_1 = a \land X_2 = b \land X_3 = c \land X_4 = d \land X_5 = e\}$

$N_6 : \{X_1 = a \land X_2 = b \land X_3 = c \land X_4 = d \land X_5 = d\}$

$X_5 \leftarrow d$

$X_5 \leftarrow e$

$N_7 : \{X_1 = a \land X_2 = b \land X_3 = c \land X_4 = d \land X_5 = e\}$

while $E_5^2$ allow us to backtrack up to $N_1$. These are however, heuristics. It is possible to come up with scenarios where picking an explanation involving constraints introduced at lower levels could have helped more.

4.3.2 Remembering and Using Learned Failure Explanations

Another open issue about this refinement search procedure is how many learned failures should be restored. Although this decision does not affect the robustness and completeness of the search, it would affect the efficiency. Specifically, there is a trade-off in the size of storage and matching costs on one-hand and search reductions on the other. Storing the failure explanations and search control rules, which are learned from all the interior nodes, could be very expensive. The CSP and machine learning literatures take different approaches to this problem.

Researchers in CSP [20][22][23] concentrated on the syntactic characteristics of the failure explanations, such as their size and minimality, to decide whether or not they should be stored. Researcher in machine learning concentrated instead on utility analysis approaches that use the distribution of the encountered problems to dynamically modify the stored rules [8]. The failure explanations learned in CSP problems have traditionally only been used in intra-problem learning, to cut down the search in the other branches of the same problems. In contrast, work in machine learning concentrated more on inter-problem learning. It becomes important to
generalize the explanations before being stored, and can thus look relevant in many situations.

5. Extending the Basic Failure-Driven Backtracking Framework for Dynamic Backtracking

The FDB framework described suffers from some possible drawbacks. In this section, we will describe those drawbacks and find ways of extending the refinement search framework to overcome them. Based on this extension, a novel algorithm which incorporates *dynamic backtracking* and *repair-based search heuristics* is proposed to solve the CSP problems efficiently.

One of the main drawbacks behind the FDB framework is that it erases all the progress made on the nodes between the ancestor node and the failure node when FDB intelligently backtracks to an ancestor node. Potentially, it might lose some useful information. From this point of view, we require an extension to FDB to provide the ability to keep intermediate work.

We here observed that we could directly traverse the space of complete assignments without deleting the values of intermediate variables during dynamic backtracking. In order to gain the flexibility from the extended refinement search framework, we might need some independent heuristics such as "min-conflict" heuristic [14].

Figure 10 presents a novel repair-based approach called *FDB-MB algorithm* that combines the advantages of the flexibility of local search with the unified refinement search framework. *FDB-MB algorithm*, based on the refinement search template, is incorporated with the spirit Min-Conflict. In FDB-MB approach, we integrate the Min-Conflict heuristic with a lazy evaluation capability, called Lazy Min-Conflict heuristic, which does not consider the unassigned value with in "sticking value" [24] marks until all other candidate variables become impossible. The tuition is that it could take some risks if the value with "sticking mark" is tried too early since its failure happened once before. It is also a complete repair-based method that works on partial and possibly inconsistent assignments. This algorithm builds a dynamically generated and simple set of failure explanations. These failure explanations are partial assignments to indicate that they violate some constraints and should be repaired.
Another feature of the FDB-MB algorithm is the ability to support incremental problem solving. The values selected and assigned to constraints naturally define a hierarchy of problem spaces. If a local inconsistency is found, then the system can backtrack to the next higher level and try to find an alternative solution. Thus the hierarchical system can be understood as a result of combining local search and

![Algorithm](https://via.placeholder.com/150)

**FIGURE 10 THE FDB-MC ALGORITHM**
backtracking problem solving. That is, in the vertical direction a backtracking method is used, while in the horizontal direction, a local repair search method is used. It could also randomly repair a defined variable that contributed to the violation of some constraints in the spirit of failure driven backtracking (FDB).

In addition, the proposed algorithm has the ability to revise solutions efficiently. An important aspect of solution revision is to maintain the stability of the revision process. It simply chooses an alternative value that minimizes the number of constraint violations and the resultant assignment does not match any failure explanations. This approach of repairing is very useful for some dynamic environment where the change of constraints is required to be minimized when new circumstances arise.

6. Relation to Existing Works

The CSP backtracking idea that is closest to the FDB formalization is the "conflict directed backjumping" (CBJ) approach proposed by Prosser [2]. This algorithm was originally proposed for binary CSP (BCSP) problems. Although the description of the whole algorithm is in terms of conflicts sets and their values, it is easy to see that conflict sets are really a stylized representation of the failure explanations for BCSP problems. The cascades of backtracking facilitated by CBJ are similar to recursive propagation. Thus, CBJ can be seen as an instantiation of FDB and a generalization of EBL of BCSPs.

In addition, a similar technique to FDB-MC algorithm has been called partial-order dynamic backtracking [1]. The motivation for this algorithm is the same as the FDB-MC algorithm, namely, allowing the search to follow local gradients in the search space while retaining the guarantee of a solution if one exists. Control of the search is done using eliminating explanation reasoning to avoid the repeated search of redundant parts of the search space. Partial-order dynamic backtracking, however, differs from the FDB-MC algorithm in that information in the form of failure explanation is removed during the search. This is an attempt to avoid the set of failure explanations becoming extremely large. A set of safety conditions or a partial ordering on the variables is generated dynamically during search and is used to assess when a failure explanation may be removed. Removing failure explanations using an operation involving such safety conditions involves extra computation. A comparison of the FDB-MC algorithm with partial-order dynamic backtracking on specific application is necessary to fully understand the differences and assess the advantages of the two approaches.
7. Summary and Conclusions

In this paper, a unified refinement search framework, which formally characterized failure driven backtracking and explanation based learning, is proposed. The main idea here is a process of explaining failure from 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 explanation of the interior nodes in other branches of the search tree.

This unified character makes CSP possible to be modeled in terms of refinement search, which helps us understand the trade-offs between the backtracking algorithms and learning techniques for CSP problems. However, there are still several issues to consider for different problem domains: (a) How the failure explanations are selected (b) How they are contextualized and represented (which involves whether or not to keep the flaw description and the description of the violated problem constraints) and (c) How to manage the storage of failure explanations.

We also presented a repair-based algorithm for solving constraint satisfaction problem based on this unified framework. The type of problems best suited to this method of search has been suggested, although much testing on specific applications is now necessary. It has been show that FDB-MC could provide advantages over systematic search. Since the algorithm also guarantees to find a solution to a CSP, if one exists, it is an improvement on most non-systematic search procedures. The flexibility and scalability of the FDB-MC algorithm must be investigated. However, the heuristics for selecting variables have the significant impact on the efficiency and scalability of the FDB-MC algorithm. Some of the variable heuristics, such as min-width and dynamic variable ordering, might be useful to select among those variables. This forms part of the future research on specific applications.

In addition, the FDB-MC algorithm, armed with repair techniques, may be well suited to dynamic CSPs, since they can be used to repair solutions that have become inconsistent due to a change in the environment. The failure explanations in the FDB-MC approach may be used to prune the search for a solution in the dynamic environment and updating the set of failure explanations is only a simple set of operation. How the FDB-MC algorithm can be used to perform and how the failure
explanations should be maintained as a dynamic support for different CSPs form a further area of research.

References

Failure-Driven Refinement Search with Local Repair-Based Heuristics for Constraint Satisfaction Problems


