Appeared in AAAI-06

Extending Dynamic Backtracking to Solve Weighted Conditional CSPs

Robert T. Effinger and Brian C. Williams

MIT – Computer Science and Artificial Intelligence Laboratory
32 Vassar Street, Bld. 32-272
Cambridge, MA 02139
{effinger,williams}@mit.edu

Abstract

Many planning and design problems can be characterized as optimal search over a constrained network of conditional choices with preferences. To draw upon the advanced methods of constraint satisfaction to solve these types of problems, many dynamic and flexible CSP variants have been proposed. One such variant is the Weighted Conditional CSP (WCCSP). So far, however, little work has been done to extend the full suite of CSP search algorithms to solve these CSP variants. In this paper, we extend Dynamic Backtracking and similar backjumping-based CSP search algorithms to solve WCCSPs by utilizing activity constraints and soft constraints in order to quickly prune infeasible and suboptimal regions of the search space. We provide experimental results on randomly generated WCCSP instances to prove these claims.

Introduction

Research on constraint satisfaction problems (CSP) has lead to many breakthroughs in our ability to understand, analyze, and solve combinatorial-style problems. These advances have taken the form of fast and sophisticated search algorithms (Dechter 1990; Ginsberg 1993; Prosser 1993), as well as in-depth complexity analyses to help differentiate between fundamentally easy and hard to solve CSP instances (Gaschnig 1979; Grant 1997). To leverage these advances into more expressive domains, such as conditional planning with preferences and design configuration, many dynamic and flexible CSP variants have emerged (Miguel 2001). One such variant, the Weighted Conditional CSP (WCCSP), employs activity constraints and soft constraints to model both conditional dependencies and preferences within a unified framework (Miguel 2001).

In this paper, we extend Dynamic Backtracking (DB) to solve WCCSPs via four extensions to Ginsberg's original algorithm (Ginsberg 1993). These extensions enable memory-bounded, conflict-directed, and optimal search of WCCSPs by utilizing activity constraints and soft constraints in order to quickly prune infeasible and suboptimal regions of the search space. While the pedagogical focus of this paper is to extend the DB algorithm in particular to solve WCCSPs, the ideas developed in this paper more generally apply to extending all backjumping-based algorithms to solve WCCSPs.

To place the ideas presented in this paper into proper perspective, we briefly review conditional, dynamic, valued and weighted CSPs, and discuss related work.

Background and Related Work

(Mittal and Falkenhainer 1990) were the first to extend the CSP to support conditional variables. This formalism is called a Conditional CSP (CCSP) (Sabin and Freuder 1999). A CCSP assumes variables enter and leave the problem according to special constraints, called *activity constraints*. It is important to point out the difference between the CCSP and the Dynamic CSP (DCSP) developed by (Dechter and Dechter 1998). The DCSP assumes that activity constraints either do not exist or are hidden from the CSP solver, thus, the addition and removal of variables appears random and unpredictable.

Valued CSPs (VCSP) incorporate soft constraints into the CSP framework to model preferences (Schiex, et. al. 1995). A Weighted CSP (WCSP) is a special instance of a VCSP in which the composition operator is constrained to be additive and non-idempotent (Shapiro and Haralick 1981; Larrosa 2002). In a WCSP, a cost is associated with each soft constraint, and the cost of a partial solution is calculated by summing the individual costs of each soft constraint violated by that partial solution.

Several CSP search algorithms have been extended to CCSPs and DCSPs. (Sabin 2003) extended arc-consistency and forward checking to CCSPs, and (Verfaillie and Schiex 1994) extended DB to DCSPs. It is important to note that the DB algorithm developed by Verfaillie and Schiex is not meant to solve CCSPs; it is unable to reason about a CCSP's activity constraints. (Miguel 2001) proposes a broad class of dynamic flexible CSP variants; the ideas presented in this paper can be viewed as filling in an empty element of Miguel's sparsely populated matrix of dynamic flexible CSP variants and algorithms. Also, (Dago and Verfaillie 1996) merge nogood learning with branch-and-bound search to make the

pruning mechanism more efficient. In this paper, we merge backjumping-based techniques with branch-and-bound search for the same desired effect. Two restricted forms of the WCCSP have been studied previously in the literature under different names. (Keppens 2002) defines the Dynamic Preference CSP in which all soft constraints must be unary constraints, and (Sabin et. al. 1995) define the Dynamic Partial CSP in which all constraints are soft and have equal cost.

The rest of this paper is organized as follows. Next, we present a formal definition of the WCCSP along with a simple motivating example. Then, we briefly review DB, and develop in detail the four extensions required to extend DB to solve WCCSPs. To conclude, we give a simple walk-through example, and provide experimental results on randomly generated WCCSP instances.

The Weighted Conditional CSP (WCCSP)

A WCCSP is a tuple, $\langle I, V, I_I, C_C, C_A, C_S, f(P)\rangle$. Where,

- $I = \{i_1, i_2, \ldots, i_n\}$, is a set of finite domain variables.
- $V = \{V_1, V_2, \ldots, V_n\}$, are finite domains for each $i \in I$.
- $I_I \subseteq I$, is a set of initially active variables.
- C_C, is a set of hard constraints that must be satisfied.
- C_A, is a set of activity constraints describing the conditions under which each variable becomes active.
- C_S, is a set of soft constraints of the form $\langle c, f_s(c)\rangle$.
- Where, c is an assignment of values to variables, and $f_s(c) \rightarrow \Re$ is a cost to be incurred if $c \in P$, where P is the current partial solution (Definition 1).
- $f(P)$, assigns a real-valued cost to a partial solution, P, by summing the costs of each soft constraint which is violated by that partial solution, $f(P) = \sum_{c \in P} f_s(c)$.

Definition 1 – **Partial solution.** A *partial solution, P,* to a WCCSP is a subset $J \subseteq I$ and an assignment of a value to each variable in J. We denote a partial solution by a tuple of pairs, (i, v) which assign the value v to the variable i. For a partial solution P, we denote by $\hat{P}$ the set of variables assigned values by P.

Definition 2 – **Activity Constraint.** An *activity constraint* is an expression of the form $AC \rightarrow active(i_k)$, where AC represents an assignment of values to variables, $\{i_1 = v_1, \ldots, i_j = v_j\}$, and is the condition under which variable i_k becomes *active*. If a variable becomes *active,* it must subsequently be assigned a value from its domain that is both consistent with the current partial solution, P, and the problem's hard constraints, C_C.

Definition 3 – **Active Variable List.** To keep track of which variables in a WCCSP are currently *active*, we define an active variable list, I_A. I_A is defined as the set of variables with satisfied activity constraint conditions, AC. Note that the set of initially active variables, I_I, will always be on the active variable list, I_A.

Definition 4 – **Solution to a WCCSP.** A *solution*, I^*, to a WCCSP is an assignment of values to all active variables, I_A, which satisfies C_C, such that $f(I^*)$ is minimized. The optimal solution to a WCCSP, I^*, is formally defined as:

$$I^* = \arg\min f(P) \quad s.t. \quad \forall c \in C_C \quad is \quad satisfied \quad and \quad \forall c \in C_A \quad is \quad satisfied.$$

We also assume that a solution is minimal with respect to activity constraints as defined in (Gelle and Faltings 2003). It is important to note that the solvability of a WCCSP depends on I_I. A solution may exist for some sets of initially active variables, I_I, but not for others.

Definition 5 – **Nogood.** A partial solution, P, is a *nogood* if and only if there is no WCCSP solution, I^*, containing P. Nogoods are commonly called conflicts.

A Motivating Example

To give a simple example of a WCCSP, we introduce a simple car configuration task commonly employed in the CCSP literature (Mittal and Falkenhainer 1990). In this example, the car buyer's objective is to minimize the cost of the vehicle subject to the car dealer's configuration requirements, shown in Figure 1. The car buyer must choose from *Base Package* (B) one of three values {*luxury, standard, convertible*}. Choosing *luxury* activates the options *Air-Conditioning* (A) and *Sunroof* (S), while

Variable	Values	Activates:	Cost $
(B) base package	1.) luxury	airConditioning, sunroof	$9 K
	2.) standard	-	$10 K
	3.) convertible	ragtop, hardtop	$9 K
(A) Air-Conditioning	1.) no	-	$0 K
	2.) yes	-	$2 K
(S) sunroof	1.) tint	-	$3 K
	2.) no tint	-	$2 K
(H) hardtop	1.) no	-	$0 K
	2.) yes	-	$2 K
(R) ragtop	1.) automatic	-	$3 K
	2.) manual	-	$2 K

Figure 1: An Example WCCSP, a car configuration task.

choosing *standard* activates no additional options, and choosing *Convertible* (C) activates the options *Hardtop* (H) and *Ragtop* (R). Each base package and option has an associated cost. Each option has a corresponding activity constraint: $C_A = \{B = 1 \rightarrow active(A),\ B = 1 \rightarrow active(S),\ B = 3 \rightarrow active(R),\ B = 3 \rightarrow active(H)\}$. In addition, the car buyer does not want a sunroof, so there are two compatibility constraints: $C_C = \{S \neq 1 \wedge S \neq 2\}$. The soft constraints in this example are all unary: $C_S = \{\{B = 1\} \rightarrow 9,\ \{B = 2\} \rightarrow 10, \{B = 3\} \rightarrow 9, \{A = 2\} \rightarrow 2, \{S = 1\} \rightarrow 3,\ \{S = 2\} \rightarrow 2, \{H = 2\} \rightarrow 2, \{R = 1\} \rightarrow 3,\ \{R = 2\} \rightarrow 2\}$.

Dynamic Backtracking (DB)

In this section, we briefly review Dynamic Backtracking (DB) as developed in (Ginsberg 1993) to solve the CSP. First, we briefly summarize the properties of DB, and then we present the DB pseudocode, with slight changes in notation, along with several requisite definitions. A more thorough development of the DB pseudocode and definitions is available in (Ginsberg 1993).

Properties of Dynamic Backtracking (DB)

DB ensures a complete, systematic, and memory-bounded search of the state space, while leveraging nogoods to only generate candidate plans that resolve all stored nogoods. When DB encounters a dead-end, it utilizes a backjumping resolution step (Proposition 1) to backjump directly to the source of the inconsistency, thus avoiding the "thrashing" behavior inherent to chronological backtracking. In addition, DB dynamically reorders the partial solution when backjumping in order to preserve as much intermediate nogood information as possible. Search failure is indicated when the backjumping resolution step returns an empty nogood, indicating that all domain values for a variable are inconsistent with the problem's constraints. DB requires $O(i^2v)$ space where i is the number of variables, and v is the largest domain size. The notation used in Proposition 1 is from (Verfaillie and Schiex 1994).

Definition 6 – **Eliminating Explanation.** Given a partial solution P to a CSP, an *eliminating explanation* for a variable i is a pair $(i \neq v, T)$ where $v \in V_i$ and $T \subseteq \hat{P}$. The intended meaning is that i cannot take the value v because of the values already assigned by P to the variables in T. An eliminating explanation can be viewed as a directed nogood. For example, if the partial solution $\{i_1 = v_1, i_2 = v_2, i_3 = v_3\}$ is a nogood, it can be written in the directed form, $\{i_1 = v_1, i_2 = v_2\} \rightarrow i_3 \neq v_3$, which corresponds to the eliminating explanation $(i_3 \neq v_3, \{i_1, i_2\})$.

Definition 7 – **Elimination Mechanism.** An *elimination mechanism* $\varepsilon(P, i)$ for a CSP is a function which accepts as input a partial solution, P, and a variable $i \notin \hat{P}$. The function returns a (possibly empty) set $E_i = \varepsilon(P, i)$ of eliminating explanations for i. An elimination mechanism tries to extend a partial solution, P, by assigning each possible value $v \in V_i$ for a variable i, and returns a reason for each value assignment that is not consistent with P. For a set E_i of eliminating explanations, Ginsberg denotes by $\hat{E}_i$ the values that have been identified as eliminated, ignoring the reasons given. Therefore, $\hat{\varepsilon}(P, i)$ returns just the values eliminated by $\varepsilon(P, i)$, while ignoring the reasons given, which is formally stated as, $\hat{E}_i = \hat{\varepsilon}(P, i)$.

Proposition 1 **- Backjumping Resolution Step.** Let i be a variable with domain, $D_i = \{v_1, v_2, \ldots, v_m\}$, and let $P_1, P_2, \ldots, P_m$ be partial solutions that do not include i. If, $P_1 \cup \{(i, v_1)\}, P_2 \cup \{(i, v_2)\}, \ldots, P_m \cup \{(i, v_m)\}$ are all nogoods, then, $P_1 \cup P_2 \cup \ldots \cup P_m$ is also a nogood.

Dynamic Backtracking Pseudocode (DB)

1. *Set* $P = E_i = \varnothing$ *for each* $i \in I$.
2. *If* $\hat{P} = I$, *return P. Otherwise, select a variable* $i \in I - \hat{P}$. *Set* $E_i = E_i \cup \varepsilon(P, i)$.
3. *Set* $L = V_i - \hat{E}_i$. *If L is nonempty, choose an element* $v \in L$ *Add* (i, v) *to P and return to step 2.*
4. *If L is empty, we must have* $\hat{E}_i = V_i$; *let* E *be the set of all variables appearing in the explanations,* T, *of each elimination explanation,* $(i \neq v, T)$ *for each* $v \in \hat{E}_i$. *(Proposition 1, Backjumping Resolution Step)*
5. *If* $E = \varnothing$, *return failure. Otherwise, let* (j, v_j) *be the last entry in P such that* $j \in E$. *Remove* (j, v_j) *from P and for each variable* $k \in P$, *which was assigned a value after j, remove from* E_k *any eliminating explanation that involves j. Set*
$$E_j = E_j \cup \varepsilon(P, j) \cup \{(j \neq v_j, E \cap \hat{P})\}$$
so that v_j *is eliminated as a value for j because of the values taken by variables in* $E \cap \hat{P}$. *Now set i = j and return to step 3.*

Extending Dynamic Backtracking to Solve Weighted Conditional CSPs (CondDB-B+B)

To extend Dynamic Backtracking to solve Weighted Conditional CSPs (CondDB-B+B), we augment the algorithm to appropriately handle activity constraints and soft constraints. This is accomplished via four extensions to the DB algorithm:

1.) A total variable ordering, I_O, for searching over conditional variables, and a conditional variable instantiation function.
2.) A modified backjumping resolution step which accounts for the behavior of conditional variables.
3.) A recursive check to remove deactivated variables from the partial solution when backjumping occurs.
4.) A branch-and-bound search framework augmented to construct minimal suboptimal nogoods.

The CondDB-B+B pseudocode is presented to the right, with each change to Ginsberg's original algorithm (DB) highlighted in grey and annotated with a superscript number indicating the extension it belongs to. Next, we give a detailed description of each of the four extensions:

Extension #1: A total variable ordering, I_O, and a conditional variable instantiation function.

One systematic method for searching over conditional variables is to construct a directed graph from the conditional dependencies between variables. Then, from this graph, a total variable ordering, I_O, can be derived which together with extensions 2 and 3 result in a systematic search over the conditional variables. This method, which we call CondBT in this paper, is described in detail in (Gelle and Faltings 2003) and (Sabin 2003).

For Extension #1, we simply merge CondBT with the DB algorithm. This extension outfits DB with the basic machinery to search systematically over conditional variables. At first glance, it may appear that CondBT's strict variable ordering strategy is incompatible with DB's dynamic variable reordering technique. However, in actuality, the two merge quite nicely for the following reasons; CondBT restricts the order in which variables are added to the partial solution, while DB restricts the order in which variables are removed from the partial solution. Therefore, the two pieces are entirely complimentary. CondBT is in charge of picking which unassigned variable should be instantiated next (in order to ensure a systematic search over the conditional variables), and DB is in charge of rearranging, reassigning, and unassigning variables once they have been instantiated (in order to perform Dynamic Backtracking).

Starting on the next page, we summarize CondBT's variable instantiation strategy in four steps:

Dynamic Backtracking for the WCCSP (CondDB-B+B)

1. *Set* $P = \varnothing$, $I = I_I$. *Set* $E_i = \varnothing$ *for each* $i \in I$. [1] *Take as input the total variable ordering,* I_O. [4] *Set the incumbent solution* $N = (\varnothing, \infty)$.

2.a. [4] *If* $\hat{P} = I_A$, *and* $f(P) < f(N)$, P *is the new incumbent solution. Set* $N = (P, f(P))$.

2.b. [4] *If* $\hat{P} = I_A$, *set* $E_i = E_i \cup (i \neq v, \hat{P} - i)$. *Otherwise, select a variable* [1] $i = applyNextAC(\ )$ *(Function 1) and set* $E_i = E_i \cup$ [4] $\varepsilon_O(P, i)$. *(Definition 8)*

3. *Set* $L = V_i - \hat{E}_i$. *If J is nonempty, choose an element* $v \in L$. *Add* (i, v) *to P and return to step 2.*

4. *If* L *is empty, we must have* $\hat{E}_i = V_i$; *let E be the set of all variables appearing in the explanations, T, of each elimination explanation,* $(i \neq v, T)$ for each $v \in \hat{E}_i$, [2] *plus all of the variables appearing in variable i's activating constraint, AC. (Proposition 2, WCCSP Backjumping Resolution Step)*

5. *If* $E = \varnothing$, [4] *return the incumbent, N. Otherwise, let* (j, v_j) *be the last entry in P such that* $j \in E$. *Remove* (j, v_j) *from P and for each variable* $k \in P$ *which was assigned a value after j, remove from* E_k *any eliminating explanation that involves j.* [3] *Call removeUnsupportedVars(j , P), and set,*

$$E_j = E_j \cup [4]\ \varepsilon_O(P, j) \cup \{(j \neq v_j, E \cap \hat{P})\}$$

so that v_j *is eliminated as a value for j because of the values taken by variables in* $E \cap \hat{P}$. *Now set i = j and return to step 3.*

[1] Extension #1, [2] Extension #2, [3] Extension #3 [4] Extension #4

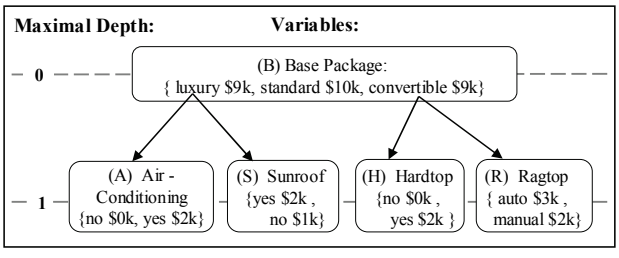

Figure 2: Dependency graph for the car buyer example.

Step 1 - Create a Dependency Graph. The dependencies between a CCSP's activity constraints, C_A, can be represented in the form of a directed graph, called a *dependency graph*, where the root node is defined as the set of all initially active variables, I_I. For example, the dependency graph for the car buyer example is shown in Figure 2, and the initially active variable is Base Package.

Step 2 – Eliminate Cycles in the Dependency Graph. Once a CCSP's dependency graph is constructed, any cycles in the graph must be eliminated by clustering the cyclic elements into a super-node. After all cycles have been collapsed, the new graph is called the *reduced dependency graph*, or RDG. The RDG will always be a directed acyclic graph (DAG). The car buyer example contain no cycles so it is trivially the RDG.

Step 3 – Derive a Total Ordering, I_O. The RDG implies a partial ordering in which the activity constraints of a CCSP should be applied and retracted during search. To determine the implied partial ordering, an integer value is defined for each node in the RDG, called the *maximal depth*. The maximal depth for each RDG node is defined as the number of nodes appearing above it in the RDG. If there happens to be more than one path into an RDG node, then the longest path must be taken as that node's maximal nest depth. For example, the nest depth of each variable in the car buyer example is shown in Figure 2, and the implied partial ordering is: $\langle\{B\},\{A,S,H,R\}\rangle$.

As defined by (Gelle and Faltings 2003), any two nodes with the same maximal depth are *incomparable,* and the order in which their corresponding activity constraints are applied is arbitrary. Thus, any total ordering, I_O, which obeys the implied partial ordering, is valid. For example, two valid total orderings for the car buyer example are $\langle B,A,S,H,R\rangle$ and $\langle B,A,S,R,H\rangle$.

Step 4 – Enforcing the Derived Total Ordring, I_O. The *CondBT* algorithm enforces a sound and complete search over the conditional variables via two additions: a total variable ordering, I_O, and a function *applyNextAC()*. The function *applyNextAC( )* works by instantiating only *active* variables, and simply skips over variables that are *not active*, and is described in Function 1.

***Function 1 - applyNextAC( ),* Conditional Variable Instantiation Function.** This function simply scans I_O from beginning to end and returns the first variable, v, which satisfies two conditions:

1.) The variable *must not* belong to the current partial solution, P. (Definition 1)
2.) The variable *must* be on the active variable list, I_A.

Extension #2: Modified Backjumping Resolution Step to account for conditional variables.

As Extension #2, we augment the Backjumping Resolution Step (Proposition 1) to account for conditional variables. To do this, we inform the backjumping resolution step that a variable may be removed from the problem via conceding any one of the activation conditions used to instantiate it. Thus, when backjumping occurs, the activation conditions responsible for a variable presently being active are also added to the newly resolved nogood. This modified resolution step is described formally below.

***Proposition 2* - WCCSP Backjumping Resolution Step**

Let i be a variable with domain, $V_i = \{v_1, v_2, \ldots, v_m\}$, activity constraint $AC \rightarrow active(i)$, and let $P_1, P_2, \ldots, P_m$ be partial solutions that do not include i. If, $P_1 \cup \{(i,v_1)\}, P_2 \cup \{(i,v_2)\}, \ldots, P_m \cup \{(i,v_m)\}$ are all nogoods, then, $P_1 \cup P_2 \cup \ldots \cup P_d \cup AC$ is also a nogood. Note that the new nogood can be resolved by removing variable i from the problem via conceding any one of its activation conditions, AC.

Extension #3: Checking for Deactivated Variables

Extension #3 is more straightforward than the previous two. When CondDB backjumps to a variable and changes its value, it is possible for that reassignment to deactivate other variables in the partial solution. In response, those variables must also be unassigned, removed from the partial solution, and the eliminating explanations depending on those variables must be erased. This is accomplished via the recursive function *removeUnsupportedVars(v,P)*.

Function 2 – removeUnsupportedVars(j , P)

for each variable $i \in \hat{P}$,

if i's activating constraint depends on the reassigned variable j, *unassign variable* i, *remove* (i,v_i) *from P, and for each variable k assigned a value after i, remove from* E_k *any eliminating explanation that involves i, call removeUnsupportedVars(i , P), and return.*

Extension #4: A Branch-and-Bound Framework Augmented with Minimal Suboptimal Nogoods

To extend DB to handle soft constraints, we integrate Branch-and-Bound (B+B) search into the DB algorithm. In addition, we augment B+B to construct minimal suboptimal nogoods. A B+B framework consists of three key attributes: an incumbent, an evaluation function, and a pruning mechanism. A basic review of B+B search is available in (Shiex et. al. 1995). For the first attribute, we simply need to initialize an incumbent, $N = (\varnothing, \infty)$, in

which we will store the lowest cost solution found so far during the search. The second attribute, an evaluation function, is already defined by the WCCSP. To incorporate the third attribute, a pruning mechanism, we need to augment DB's elimination mechanism, ε, to prune suboptimal partial solutions as well as inconsistent ones. This is described formally in Definition 8. Additionally, we augment DB's elimination mechanism to construct minimal suboptimal nogoods, as described in Function 3.

***Definition 8* - WCCSP Elimination Mechanism, $\varepsilon_O(P,i)$.** We define a new *elimination mechanism* ε_O for the WCCSP as a function which accepts as arguments a partial solution, P, and a variable $i \notin \hat{P}$ and returns a (possibly empty) set $E_i = \varepsilon_O(P,i)$ of eliminating explanations for i. An elimination mechanism tries to extend a partial solution, P, by assigning each possible value for a variable i, and returns a reason for each value assignment which along with P is inconsistent or suboptimal given the problems constraints, C_C and C_S. If the extended partial solution is suboptimal, (has a cost greater than the current incumbent), $f(P \cup i = v) \geq f(N)$, then a subset of the extended partial solution, $M \subseteq P \cup i$ is returned as the reason for inconsistency, since its extension will be a *suboptimal* solution, $(i \neq v, \hat{M})$. Where M is determined by the function *minSubOptimalNogood(P,f(N))*.

Function 3 – minSubOptimalNogood(P , f(N))

Let $C_E \subseteq C_S$ be the set of soft constraints expressed in the partial solution P, and thus contributors to $f(P)$. Let $\langle C_E \rangle$ be an ordered list of each constraint $c \in C_E$ such the associated costs, $f_s(c)$, are ordered from greatest to least. Let C_k be the first k elements of $\langle C_E \rangle$ such that their combined cost exceeds $f(N)$. Let M be the set of all variable assignment from each $c \in C_k$. Return M as a minimal suboptimal nogood.

This concludes our description of the four extensions to DB in order to solve WCCSPs. Proofs of completeness and termination of CondDB-B+B are sketched out in the thesis summarized by this paper (Effinger 2006).

Taking CondDB-B+B for a Test-Drive

In this section we give an execution trace of the CondDB-B+B algorithm solving the car buyer example. Initially, we assume *CondDB-B+B* receives as input, $I_I = \langle B \rangle$ and $I_O = \langle B, A, S, H, R \rangle$. In Figure 3, we show at each search step the partial solution, P, its cost, $f(P)$, the cost of the current incumbent, $f(N)$, the elimination explanations for each variable, E_i, and also the set of active variables which are not assigned values, $I_A - \hat{P}$.

Search Step	P	f(P)	f(N)	$I_A - \hat{P}$	E_i	B	A	S	H	R
0	{ ∅ }	-	∞	{ B }	1					
					2					
					3					
1	{B=1}	9	∞	{ A,S }	1	✓				
					2					
					3					
2	{B=1, A=1}	9	∞	{ S }	1	✓	✓			
					2					
					3					
3	{B=1, A=1, ... }	9	∞	{∅}	1	✓	✓	{∅}		
					2			{∅}		
					3					
4	{B=2}	10	10	{∅}	1	{∅}				
					2	☑				
					3					
5	{B=3}	9	10	{ H,R }	1	{∅}				
					2	{∅}				
					3	✓				
6	{B=3, H=1}	9	10	{ R }	1	{∅}			✓	
					2	{∅}			{B}	
					3	✓				
7	{B=3, H=1, ... }	9	10	{∅}	1	{∅}			✓	{B}
					2	{∅}			{B}	{B}
					3	✓				
8	{B=∅}	9	10	{ H,R }	1	{∅}				
					2	{∅}				
					3	{∅}				
Return the Incumbent Solution, N = ({B=2} , 10).										

Figure 3: Solving the car buyer WCCSP with CondDB-B+B

Step 0 – *CondDB-B+B* is initialized with $P = \varnothing$, $\hat{E}_i = \varnothing$, $I_A = I_I = \langle B \rangle$, $I_O = \langle B, A, S, H, R \rangle$, and $N = (\varnothing, \infty)$.

Step 1 - $\hat{P} \neq I_A$, so the function *applyNextAC( )* is called and returns variable B. All three of $B's$ value assignments are consistent with the constraints, C_C, and it is assigned the value 1. The new variable-value assignment $B = 1$ activates variables A and S.

Step 2 - $\hat{P} \neq I_A$, so *applyNextAC( )* is called and returns variable A. Both of $A's$ value assignments are consistent with the constraints C_C, and it is assigned the value 1.

Step 3 - $\hat{P} \neq I_A$, so *applyNextAC( )* is called and returns variable S. Both of $S's$ value assignments are *NOT* consistent with the constraints, C_C, so it is not added to the partial solution. The buyer does not like a sunroof! Since S is self-inconsistent, each of its elimination explanations are empty, $\{\varnothing\}$. $E_S = E_S \cup (S \neq 1, \{\varnothing\}) \cup (S \neq 2, \{\varnothing\})$.

Step 4 – $L = \varnothing$, so backjump, $E = \{\varnothing \cup \varnothing \cup B\} = \{B\}$. (Proposition 2) Let $(j, v_j) = (B,1)$. Remove $(B,1)$ from P, and erase any elimination explanations involving *B*. Call *removeUnsupportedVars(B,P),* which removes A from P. Set $E_B = E_B \cup (B \neq 1, \{\varnothing\})$, and set $B = 2$. Now, $\hat{P} = I_A$ and $f(P) < f(N)$, so that P is the new incumbent. Set $N = (\{B = 2\}, 10)$.

Step 5 - $\hat{P} = I_A$ so $E_B = E_B \cup (B \neq 2, \hat{P} - B)$ and $(B,3)$ is added to P. The assignment $\{B = 3\}$ activates *H* and *R*.

Step 6 - $\hat{P} \neq I_A$, so *applyNextAC( )* is called and returns variable H. The assignment $H = 2$ is pruned as suboptimal, $f(P \cup \{H = 2\}) \geq f(N)$. Add $H = 2$ to P.

Step 7 - $\hat{P} \neq I_A$, so *applyNextAC( )* is called and returns variable R. Both of $R's$ value assignments are pruned as suboptimal. Note that the minimization function (Def. 14) eliminates H=1 from each no-good, since the cost of the assignments to B and R alone (without even considering H) already comprise suboptimal nogoods.

Step 8 - $L = \varnothing$, so backjump, $E = \{\varnothing\}$. An empty nogood was produced. Return the incumbent, $N = (\{B = 2\}, 10)$. Search Success!!

Experimental Results

To test the CondDB-B+B algorithm, we developed a random WCCSP generator which accepts three inputs: the number of desired variables, the maximum domain size, and the maximum depth of nested activity constraints. We tweaked the random generator to output WCCSP instances that lie near the phase transition by varying the ratio of binary hard constraints to variables until the generated instances were approximately 50% solvable. The generator randomly constructs one activity constraint for each variable, and one soft constraint per variable-value assignment with a uniform cost distribution from 1 to 10.

We benchmarked the CondDB-B+B algorithm against a standard branch-and-bound algorithm augmented to handle conditional variables (CondBT-B+B). We performed two separate experiments. For the first experiment, the domain size was fixed at 3, the maximum depth of activity constraints was fixed at 4, and the number of variables was varied from 6 to 20. For the second test, the domain size was fixed at 3, and the depth of activity constraints, *n*, was varied from 1 to 6. The activity constraints formed a uniform depth tree, so that the number of variables and the number of activity constraints grew at each step by approximately 3^(*n*) as *n* varied from 1 to 6. As the metric for comparison, we counted the number of search tree nodes tested before an optimal solution or search failure was returned. For each data point, we tested 100 WCCSP instances, and individual tests were capped at 5000 candidates. The results, presented in Figures 4 thru 7, show that CondDB-B+B provides a significant improvement in average and worst-case case performance along two WCCSP

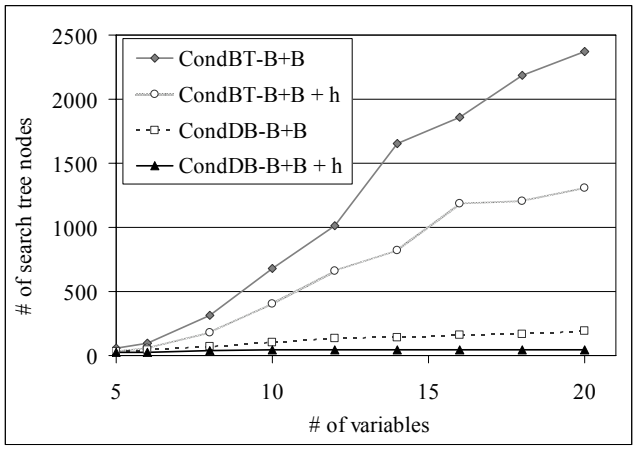

Figure 4: Average Case Test Results - Fixed Depth of Nested Activity Constraints

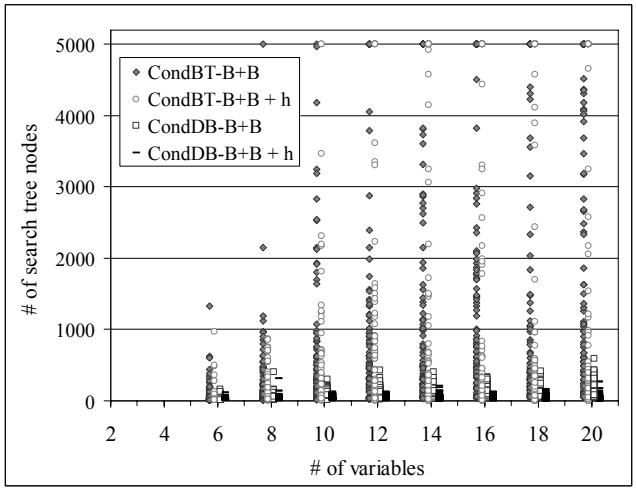

Figure 5: Individual Test Case Results - Fixed Depth

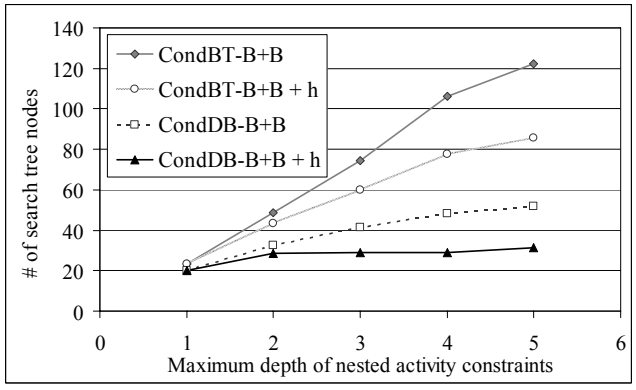

Figure 6: Average Case Test Results – Varying Depth of Nested Activity Constraints

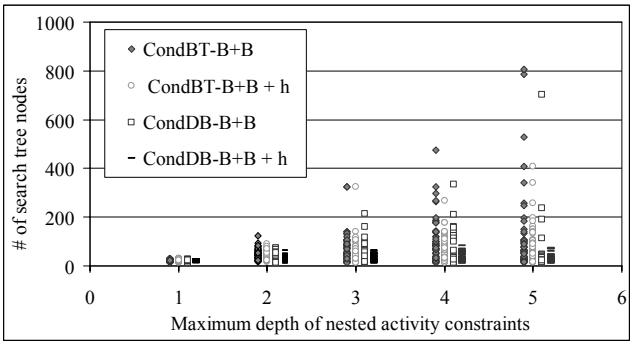

Figure 7: Individual Test Case Results - Varying Depth

dimensions; the number of variables and the maximum depth of nested activity constraints. These promising results motivate more extensive testing, and imply that DB and similar backjumping-based algorithms can continue to increase search efficiency in extended CSP domains, such as WCCSPs, by using nogoods to avoid thrashing.

Future Work

One promising direction for future work is to extend more sophisticated optimal search techniques, such as Russian Doll Search (Verfaillie et. al. 1996) to WCCSPs. For example, when a WCCSP's dependency graph is strictly a tree, the cost of instantiating a sub-tree can be directly attributed to the parent assignment which activates it. This relationship can be used as a tighter upper bound when B+B prunes based on suboptimality. Preliminary results show that even this simple heuristic can significantly improve performance as indicated in Figures 4 thru 7 with the names *CondBT-B+B+h* and *CondDB-B+B+h.* Local consistency checking and nogood recording could also be incorporated into CondDB-B+B to improve its performance, and an A* style search could potentially improve performance at the cost of more memory.

Acknowledgements

We thank the anonymous reviewers for their thoughtful comments. This research was supported by the NASA H&RT program under contract NNA04CK91A, and the DARPA SRS program under contract FA8750-04-2-0243.

References

Dago, P. and Verfaillie, G. 1996. Nogood Recording for Valued CSPs. In *Proc. of ICTAI-96*.

Dechter, R. 1990. Enhancement Schemes for Constraint Processing: Backjumping, Learning and Cutset Decomposition. *Artificial Intelligence*, 41(3):273–312.

Dechter, R., and Dechter, A. 1988. Belief Maintenance in Dynamic Constraint Networks. *In AAAI '88*, 37-42.

Effinger, R. 2006. Optimal Temporal Planning at Reactive Time Scales via Dynamic Backtracking Branch-and-Bound. S..M. diss., MIT.

Gaschnig, J. 1979. Performance Measurement and Analysis of Certain Search Algorithms, Tech. Rept. CMU-CS-79-124, Carnegie-Mellon University, Pittsburgh, PA.

Gelle, E. and Faltings, B. 2003. Solving mixed and conditional constraint satisfaction problems. *Constraints,* 8(2):107–141.

Ginsberg, M. 1993. Dynamic Backtracking. *Journal of Artificial Intelligence Research* 1:25--46.

Grant, S. 1997. Phase Transition Behaviour in Constraint Satisfaction Problems. Ph.D. diss., The Univ. of Leeds.

Keppens, J., 2002. Compositional Ecological Modelling via Dynamic Constraint Satisfaction with Order-of-Magnitude Preferences. Ph.D. Thesis. Univ. of Edinburgh.

Larrosa., J. 2002. On Arc and Node Consistency in Weighted CSP. In *Proc. AAAI'02*, Edmondton, CA.

Miguel, I. 2001. Dynamic Flexible Constraint Satisfaction and Its Application to AI Planning. PhD diss., The Univ. of Edinburgh.

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

Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problems. *Comp. Intelligence*, 9(3):268-299.

Sabin D., et. al. 1995. A Constraint-Based Approach to Diagnosing Configuration Problems. *In Proc. IJCAI-95. Workshop on AI in Distributed Information Networks.*

Sabin, M. 2003. Towards More Efficient Solution of Conditional CSPs. Ph.D. diss. The Univ. of New Hampshire.

Sabin, M., and Freuder, E. 1999. Detecting and resolving inconsistency in conditional constraint satisfaction problems. In *AAAI'99 Workshop on Configuration*, 95-100.

Schiex, T., et. al. 1995. Valued constraint satisfaction problems. In *Proceedings of IJCAI'95*, 631-637.

Shapiro, L. and Haralick, R. 1981. Structural descriptions and inexact matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3:504–519.

Verfaillie, G., Lemaitre, M., T. Schiex. 1996. Russian Doll Search, *In Proc. of AAAI'96*. Portland, OR.

Verfaillie, G. and Schiex, T., 1994. Dynamic Backtracking for Dynamic CSPs. In *ECAI'94*.