Partial Order Backtracking

David A. McAllester
MIT Artificial Intelligence Laboratory
545 Technology Square
Cambridge Mass. 02139
dam@ai.mit.edu

Abstract: Ginsberg has recently introduced a polynomial space aggressive dependency directed backtrack search technique. Aggressive dependency directed techniques have the property that the size of the search space generated by a union of disjoint subproblems is proportional to the sum of the search space generated by each problem independently. Earlier polynomial space dependency directed methods did not have this property. This paper presents two additional polynomial space aggressive dependency directed algorithms that allow greater flexibility in the use of variable and value ordering heuristics. One of the most important heuristics in constraint satisfaction search involves dynamic rearrangement of the order in which variables are considered. The second procedure presented here, which we call partial order backtracking, allows the order of past variables to be dynamically improved during the search.

1 Introduction

We consider search techniques for finding solutions to constraint satisfaction problems (CSPs). A CSP consists of a set of variables, each with an associated domain (set of possible values), and a set of constraints. Dependency directed backtracking, also known as intelligent backtracking, reduces the search space of classical backtrack search programs by recording "dependencies" or "justifications" at failed nodes in the search tree. At any point during a classical search some of the variables have been assigned values and others remain unassigned. The assigned variables are often called "past" variables and the unassigned variables are called "future" variables. Most dependency directed backtracking techniques can be divided into one of two types that I will call aggressive and conservative. Conservative techniques, such as those described in [Gaschnig, 1979] and [Bruynooghe and Pereira, 1984], only store dependency information for previously tried values of the past variables. Aggressive techniques, such as that described in [Stallman and Sussman, 1977], keep dependency information about future variables derived from earlier portions of the search involving those variables. Most aggressive techniques are impractical for many applications because they require exponential space to store all the dependency information which accumulates during a search. Conservative techniques require only polynomial space to store the dependency information and can be implemented with little overhead relative to classical backtracking. However, conservative techniques achieve a smaller degree of search tree pruning than that achieved by aggressive techniques. Ginsberg has recently introduced a dependency directed search technique requiring only polynomial space and yet which is aggressive in the sense that it maintains useful dependency information about future variables. Ginsberg has presented evidence that this technique is useful in solving crossword puzzles [Ginsberg *et al.*, 1990] [Ginsberg, 1993].

Although aggressive dependency directed techniques can achieve useful search tree pruning in a variety of types CSPs, the advantages of aggressive techniques are perhaps best illustrated by considering a CSP which is a union of disjoint subproblems. When presented with such a CSP traditional backtracking procedures can require time proportional to the product of sizes of the search spaces of the two subproblems. To see this consider the case where one subproblem is solvable but the other is not and the variables of the solvable subproblem are considered before the variables of the unsolvable subproblem. Classical backtracking will find a solution for the first problem and then search for a solution for the second. The search for the second solution will fail and backtracking will cause the search on the second subproblem to be repeated if a second solution for the first subproblem is found. In fact the second subproblem will be searched repeatedly for each solution to the first subproblem. A similar analysis holds if the variables of the two subproblems are considered in some interleaved order. In this case search can take time proportional to the product of the sizes of the two search spaces even if both subproblems are solvable. Conservative dependency directed techniques behave in a manner similar to classical backtracking on these examples. Ginsberg's algorithm, and the two generalizations presented here, solve a union of disjoint subproblems using a search no larger than the sum of the sizes of the search spaces of the independent subproblems. More concisely, we can say that the algorithms are additive on disjoint unions.

The poor performance of backtrack search on unions of disjoint subproblems has been observed earlier and several procedures exist for splitting CSPs into independent subproblems which can then be solved separately, e.g., [Seidel, 1981]. However, Ginsberg's algorithm is motivated by a larger class of CSPs in which nogoods on future variables have a significant impact on search size. As mentioned above, Ginsberg has given empirical evidence for the value of dependency information on future variables in the case of crossword puzzles. To our knowledge the behavior of Seidel's techniques on crossword puzzles has not been empirically investigated but it seems unlikely to do well because it requires selecting the variables in such a way that unconstrained variables are selected early. This is counter to the empirically important heuristic of selecting highly constrained and constraining variables first [Purdom and Robinson, 1981], [Zabih and McAllester, 1988], [Crawford and Auton, 1993].

This paper does not contain empirical data. It is intended as a theoretical presentation and analysis of two new polynomial space aggressive dependency directed algorithms. The first procedure is a strict generalization of Ginsberg's algorithm in the sense that Ginsberg's algorithm corresponds to a restriction on the heuristic choices available in the more general algorithm. The second algorithm is not a generalization of Ginsberg's technique in this sense but is theoretically motivated in two ways. First, it allows the ordering of past variables to be dynamically improved during the search. Second, it allows some use of "heuristic repair" or "gradient descent" methods as described in [Minton *et al.*, 1990], [Sosič and Gu, 1991], and [Selman *et al.*, 1992] to be used in a systematic search procedure.

2 Basic Concepts

We begin by establishing the basic framework of constraint satisfaction problems (CSPs).

> **Definition:** A *constraint satisfaction problem* (CSP) is a finite set of variables and a finite set of constraints where each variable x has an associated finite set D_x called the domain of x and each constraint is an expression of the form $P(x_1, \ldots, x_n)$ where each x_i is a variable of the CSP and P is a predicate on those variables, i.e., a subset of $D_{x_1} \times \ldots \times D_{x_n}$. A *solution* to a CSP is an assignment ρ of values to variables such that for each variable x we have that $\rho(x) \in D_x$ and for each constraint $P(x_1, \ldots, x_n)$ we have $<\rho(x_1), \ldots, \rho(x_n)> \in P$.

A Boolean satisfiability problem can be viewed as a CSP with Boolean variables and disjunctive constraints. It follows that determining whether a given CSP has a solution is an NP complete problem. A variety of heuristic methods for improving the efficiency of procedures for finding solutions to CSPs have been proposed. Here we are primarily concerned with heuristic techniques based on "dependency analysis". When a classical backtrack search procedure determines that a given partial assignment can not be extended to a solution it is often possible to find a subset of the given partial assignment which is "responsible" for the failure. For example, in a partial assignment assigning a value to ten variables there may be a subset of three "culprit" variables such that the restriction of the assignment to just the culprit variables has no extension to a solution. A nogood is a data structure which records a partial assignment that can not be extended to a solution.

> **Definition:** A nogood is a Boolean formula of the form
>
> $$\neg(x_1 = v_1 \ \wedge \ x_2 = v_2 \ \wedge \ldots \wedge x_n = v_n)$$
>
> where each x_i is a variable and $v_i \in D_{x_i}$. An assignment of values to variables *violates* a nogood if it assigns each x_i the value v_i. A nogood *holds* in a CSP if no solution to that CSP violates the nogood. The conjunction in the nogood can be empty resulting in the *empty nogood.* The empty nogood is universally false, i.e., it is violated by all truth assignments.

The nogood $\neg(x_1 = v_1 \ \wedge \ x_2 = v_2 \ \wedge \ldots \wedge x_n = v_n)$ is logically equivalent to the implication $(x_1 = v_1 \ \wedge \ x_2 = v_2 \ \wedge \ldots \wedge \ x_{n-1} = v_{n-1}) \rightarrow x_n \neq v_n$. We have the following inference rules for deriving nogoods that hold in a given CSP.

1. Derive $\neg(x_1 = v_1 \ \wedge \ x_2 = v_2 \ \wedge \ \ldots \wedge x_n = v_n)$ whenever this nogood is implied by a single constraint of the CSP.

2. If $w_1, \ldots, w_n$ are the possible values of y then we have the rule

$$\frac{\begin{array}{l}\Sigma_1 \rightarrow y \neq w_1 \\ \vdots \\ \Sigma_n \rightarrow y \neq w_n\end{array}}{\neg(\Sigma_1 \wedge \ldots \wedge \Sigma_n)}$$

Note that rule 2 can be used to derive an empty nogood. It is possible to show that a given CSP is satisfiable (has a solution) if and only if the above rules can not be used to derive the empty nogood. In particular, the generalized dynamic backtracking procedure given below either finds a solution or derives the empty nogood.

3 Generalized Dynamic Backtracking

We are interested in constructing a backtrack procedure which saves useful valid nogoods about future variables and yet uses only polynomial space. To ensure the utility of nogoods about future variables the procedure maintains a *total* assignment of values to variables. At most times during the search this total assignment will violate some constraints (if it does not then a solution has been found). However, the total assignment is used to preserve the state of the "future" part of the search — it stores the state of the search at the time of backtracking. In order to maintain a polynomial upper bound on the number of nogoods we only store nogoods that are relevant to the current total assignment. The notion of relevance is defined below. The procedure also maintains a total order on variables which intuitively corresponds to the order used to distinguish past from future variables. Note that since we have a total assignment of values it is not the case the future variables are unassigned. The variable order can change during the search. Again, to maintain a polynomial upper bound on the number of nogoods while preserving a useful set of nogoods we require that each nogood respect the total order.

> **Definition:** Let $(x_1 = v_1 \wedge x_2 = v_2 \wedge \ldots \wedge x_n = v_n) \rightarrow z \neq w$ be a nogood. The nogood is said to *respect* an ordering $<$ on variables if for each x_i we have $x_i < z$. The nogood is said to be *relevant* to an assignment ρ of values to variables if for each x_i we have $\rho(x_i) = v_i$.

We will also use the following notation.

> **Definition:** For any assignment ρ we define $\rho[x := v]$ to be the assignment identical to ρ except that it maps x to v. We define $\rho|_{<,x}$ to be the partial assignment that is the restriction of ρ to those variables less than or equal to x under the order $<$.

The generalized dynamic backtracking procedure **GDB** given below takes a single argument i which is an integer between one and the number of variables in the CSP. The variables appearing earlier than the ith position under the total order $<$ will be called the past variables and variables at position i or latter are called the future variables. When the procedure is called the restriction of the assignment ρ to past variables does not violate any constraints or nogoods. The procedure only alters the variable assignment and the total order on future variables and it either finds a solution or generates a new nogood involving only past variables. Initially the total order $<$ and assignment ρ are arbitrary and the nogood set Γ is empty. Calling **GDB**(1) either generates a solution or the empty nogood.

Procedure GDB(i):

Until ρ is a solution or a new nogood has been generated involving only variables prior to the ith variable do the following:

1. Select a variable x at position i or later under the order $<$. Change the ordering so that x is now the ith variable under the new ordering and delete all nogoods in Γ that do not respect the new ordering.

2. If there is some value v such that $\rho[x := v]|_{<,x}$ does not violate any constraint or nogood then select such a value v, change the current assignment to $\rho[x := v]$, remove all nogoods not relevant to $\rho[x := v]$, and call **GDB**$(i + 1)$.

3. If there is now (after step 2) no value v such that $\rho[x := v]|_{<,x}$ does not violate any constraints or nogoods, then it must be possible to combine some number of applications of rule 1 with a single use of rule 2 to derive a new nogood involving only variables prior to x under the order $<$. Derive such a nogood and return.

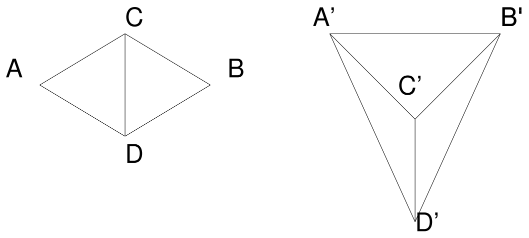

Figure 1: A Union of Disjoint Graph Coloring Problems

The above procedure is designed to satisfy three conditions none of which are obvious. First, for most reasonable heuristics for selecting variables and values in steps 1 and 2 it is additive on disjoint unions. Second, it is systematic in the sense that the number of leaves in the search tree is bounded by the product of the domain sizes of the variables. Third, it uses only polynomial space.

Ginsberg's dynamic backtracking procedure also satisfies the three conditions just mentioned. The above procedure differs from Ginsberg's in the flexibility of the choice of variable at step 1. Ginsberg's procedure maintains both assigned and unassigned future variables and is only free to make a choice in step 1 when all future variables are unassigned. The flexibility of the choice at step 1 is important because it allows the use of a variety of dynamic variable ordering heuristics which have been shown to be important in practice, e.g., [Zabih and McAllester, 1988] [Crawford and Auton, 1993].

As an example consider the graph coloring problem shown in figure 1. The figure shows two independent graphs. We are interested in finding a coloring of the union of these graphs using three colors. The first graph is three colorable but the second graph is not. Suppose that the selection of variable at step 1 of the **GDB** procedure always selects the variables in the order A, A', B, B', C, C', D, and D'. In other words, we interleave the variables of the two disjoint subproblems. Furthermore, suppose that we represent the four colors with integers and that step 2 of the GDB procedure always selects the smallest integer possible. Under this strategy for selecting variables and values each addition of a nogood makes progress in one of the two subproblems without disturbing the nogoods or assignment being used for the other subproblem. This is because nogoods for the first problem are not removed while backtracking in the second problem and vice versa. In most polynomial space dependency directed backtracking techniques a nogood is removed whenever one backtracks over the conclusion variable. Because aggressive dependency directed techniques keep nogoods on future variables, the needed nogoods are retained.

It should be noted that GDB is not necessarily additive on disjoint subproblems. In particular, if the choices in steps 1 and 2 are randomized then backtracking in one subproblem can cause the variable order and value assignment in the other subproblem to be randomized. In this case backtracking in one subproblem can remove nogoods in other subproblem. However, under most reasonable deterministic heuristics for selecting variables and values at steps 1 and 2, the two subproblems will behave independently.

I will now show that under any execution of the above procedure the number of leaves in the call tree is no larger than the product of the domain sizes of the variables. The call tree is defined to be the tree whose nodes correspond to calls to **GDB** and where one node is a child of another if the former is a call in the execution of the latter. We define the domain size of a variable x relative to position i to be number of elements of D_x which are not rules out by nogoods whose antecedents mention only variables at positions earlier than i. Intuitively, the domain size of x relative to position i is the number of values of x that are "viable" during the execution of $\mathbf{GDB}(i)$. Recall that on a given call to **GDB** variables at positions earlier than i are called *past* variables and variables at position i or earlier are called *future* variables. It is easy to see that the procedure terminates since every recursive call involves a larger value for i and each time around the loop the sum of the domain sizes (relative to i) of the future variables is reduced. We show that the number of leaves in the execution tree of the computation of $\mathbf{GDB}(i)$ is never larger than the product of the domain sizes, relative to i, of the future variables. The procedure maintains the invariant that at the beginning of the loop all future variables have a domain size (relative to i) of at least one. We now proceed by induction on the product of the domain sizes of the future variables. If the product of the domain sizes of future variables is exactly one then the value of every future variable is "fixed" by the current values of the past variables. In this case the computation for $\mathbf{GDB}(i)$ involves at most one recursive call to $\mathbf{GDB}(i+1)$ and there is only a single leaf node in the call tree. To prove the inductive case suppose we have selected the variable x at step 1 of the procedure. Let d be the domain size of x (relative to i) and let k be the product of the domain sizes (relative to i) of the future variables other than x. Now consider a call to $\mathbf{GDB}(i+1)$ generated by the current call. By the induction hypothesis the call graph of this recursive call can have at most k leaf nodes. When it returns either a nogood involving only past variables is generated, in which case the current call returns and we are done, or the domain size of x is reduced by 1. In the latter case the call tree of the remainder of the computation can have at most $(d-1)k$ leaves. So the total number of leaves is at most dk which is the product of the domain sizes of future variables.

Finally we show that the number of nogoods never climbs above nd where n is the number of variables and d is the maximum domain size. Each nogood has a distinguished conclusion of the form $x \neq v$. It suffices to show that no two nogoods have the same conclusion. To do this one must show that when a new nogood is added in step 3 of the procedure there is currently no nogood with the same conclusion as the newly generated one. Suppose there was such a nogood. Since all nogoods must be relevant to the current assignment, that earlier nogood must rule out a value of an earlier variable. But, since all nogoods must be relevant, the newly generated nogood must rule out the *current* value of some previous variable. If the new and old nogood share the same conclusion then the old nogood must rule out the current value of some previous variable. But this contradicts the invariant that the restriction of the current assignment to earlier variables does not violate any of the earlier nogoods.

4 Partial Order Backtracking

In this section we present another polynomial space aggressive dependency directed backtrack procedure. This procedure maintains only a partial order on variables and we call it partial order backtracking (POB). The use of a partial order on variables introduces a great deal of flexibility into the search. This flexibility has two motivations. The first motivation involves the effective use of variable ordering heuristics. It is well known that the selection of variable ordering is heuristically important in backtrack search. By using a least commitment approach to variable ordering one can dynamically improve the order of past variables during the search. The second motivation for partial order backtracking involves hill climbing heuristics for finding solutions more quickly in those cases where solutions exist. Recently there has been considerable interest in hill climbing or local search techniques for solving CSPs [Minton *et al.*, 1990], [Sosič and Gu, 1991], [Selman *et al.*, 1992]. Although it seems unlikely that any systematic procedure can exploit the full power of hill climbing techniques, it may be possible to reduce the time required to find solutions by incorporating some relatively restricted hill climbing heuristics into systematic search. Partial order search makes this possible by working with a total assignment of values to variables and by allowing considerable freedom in the modification of this assignment as the search progresses. This is one of Ginsberg's original motivations for dynamic backtracking.

The partial order backtracking procedure maintains three data structures — a total assignment ρ of values to variables, a set Γ of nogoods, and a set of *safety conditions* each of which is an expression of the form $x < y$ where x and y are variables of the CSP being solved. A safety condition of the form $x < y$ intuitively expresses the statement that x precedes y in the variable ordering being used at a certain point in the search. Each nogood is represented as an implication with a distinguished conclusion.

> **Definition:** A *topological sort* of set Σ of safety conditions and a set Γ of nogoods is a total order s on the variables such that for each safety condition $x < y$ in Σ we have that x precedes y in s and for each nogood in Γ we have the each variable in an antecedent of the nogood precedes the variable in the conclusion.

Each state of the computation has a set of nogoods and safety conditions. A topological sort of those nogoods and safety conditions will be called a topological sort of the order constraints at that state of the computation. If a variable x precedes a variable y in all topological sorts of the order constraints then we say that x *must* precede y under those constraints. If there exists a topological sort in which x precedes y then we say that x *may* precede y.

> **Definition:** The *live domain* of a variable z relative to a set Γ of nogoods is the set of values v in D_z such that $z \neq v$ is not in the conclusion of any nogood in Γ.

The procedure POB uses the auxiliary backtrack procedure given below. The backtrack procedure either derives the empty nogood or adds new nogoods while preserving the invariant that the value assignment ρ satisfies all the nogoods in Γ. The backtrack procedure takes a single new nogood as an argument. The procedure is only given nogoods that violate the current assignment. A call to the backtrack procedure to add a new nogood is analogous to the backtracking phase of a classical backtrack search procedure. A call to this procedure will be called a *backtrack event.* Each nogood added in a backtrack event reduces the live domain of the variable in the conclusion of the nogood. The procedure either changes the value of the variable in the conclusion of the nogood or propagates failure to earlier variables. In any case, a backtrack event either backtracks all the way to the "root" of the search and generates the empty nogood or produces a state in which the assignment ρ is consistent with the nogood set Γ.

Procedure for backtracking from the nogood $\neg(x_1 = v_1 \ \wedge \ x_2 = v_2 \ \wedge \ldots \wedge \ x_n = v_n)$

1. If the given nogood is the empty nogood then return.
2. Represent the nogood as an implication $(y_1 = v_1 \wedge \ldots \wedge y_n = v_n \rightarrow z \neq w)$ in a way that is consistent with the current order constraints, i.e., so that there exists a topological sort of the order constraints in which each y_i precedes z.
3. Add the nogood (represented as an implication) to the set Γ.
4. If the live domain of z is now empty then it must be possible to apply the second inference rule for deriving new nogoods to derive a nogood involving variables constrained to precede z. In this case derive such a nogood and recursively backtrack from it. If the recursive backtrack generates the empty nogood then return.
5. If no new nogood was generated in step 4, i.e., if the nogood added in step 3 did not eliminate all live values of z, then for each variable w which must follow z under the current order constraints add the safety condition $z < w$ and delete all safety conditions of the form $y < w$.
6. The assignment ρ is now consistent with any nogoods added in step 4. This implies that the live domain for z must now be nonempty. Set ρ to $\rho[z := w]$ where w is in the current live domain of z and remove all nogoods not relevant to the new assignment.

We can now define partial order backtracking.

> **Procedure POB:** Until ρ is a solution or the empty nogood has been derived select a nogood implied by the constraints and violated by the current assignment and backtrack from that nogood.

We now give a bound on the running time of the procedure, show that it uses only polynomial space, and show that it is additive on disjoint unions. First consider a single call to the backtrack procedure. The recursive calls to the backtrack procedure must terminate because each recursive call involves variables earlier in the partial order. The partial order is extended monotonically as each nogood is added. This monotonic extension preserves the consistency of the partial order, i.e., it preserves the property that there exists at least one topological sort of the order constraints. Since each recursion involves earlier variables the number of recursive calls can not exceed the number of variables. We show below that the number of backtrack events (calls to the nogood addition procedure) can not exceed d^n where n is the number of variables and d is the maximum domain size of any variable. To show that the procedure uses only polynomial space we note that the procedure maintains the invariant that no two nogoods have the same conclusion. This implies that there are at most nd nogoods where n is the number of variables and d is the maximum domain size. Finally, consider a CSP which is a union of two disjoint subproblems. Each backtrack event involves only variables in one of the two subproblems. Therefore the partial order backtrack procedure clearly solves a union of disjoint subproblems in time proportional to the sum of the time required to solve each subproblem independently.

To prove that the number of backtrack events (calls to the nogood addition procedure) can not exceed d^n where n is the number of variables and d is the maximum domain size of any variable we associate each state of the computation with a vector of numbers which decreases lexicographically at each backtrack event. Define $|x|_\Gamma$ to be the size of the live domain of the variable x relative to the nogood set Γ. If $<x_1, \ldots, x_n>$ is a topological sort of the order constraints at some state in the procedure, then $<|x_1|_\Gamma, \ldots, |x_n|_\Gamma>$ will be called the domain size tuple of that topological sort. The lexicographically *largest* domain size tuple of all topological sorts of the partial order of a given state of the procedure will be called the *maximal domain size tuple* of that state. We show below that each backtrack event lexicographically decreases the maximal domain size tuple. Clearly a tuple of n numbers between 1 and d can not decrease lexicographically more than d^n times. Because of the complexity of the argument, the proof that the maximal domain size tuple decreases at each backtrack event is postponed until the end of this section.

We expect the performance of the partial order backtrack procedure to be sensitive to the variety of nondeterministic choices that are made for each backtrack event. First, there is the choice of which clause to pass to the top level call of the backtrack procedure. Second, there is the choice of the conclusion variable in step 2. Third, there is the choice of value in step 6. The phrase "backtrack event" refers to a particular selection for all of the choices involved in a single top level call to the backtrack procedure. We give two (noneffective) heuristics for selecting backtrack events.

Acceleration Heuristic: Define the *minimal domain size tuple* of a state of the computation to be the lexicographical minimum of all domain size tuples over all topological sorts of the order constraints. Select a backtrack event which minimizes the minimal domain size tuple of the resulting state.

Repair Heuristic: Select a backtrack event such that the resulting assignment ρ violates the minimum number of constraints of the CSP.

Neither of the above two heuristics can be implemented directly because no method has been given for finding backtrack events with the desired properties. However, the above heuristics can be used as guidelines in constructing more practical heuristics. For example, one can approximate the acceleration heuristic by maintaining at all times a particular topological sort of the variables. The choice of nogood conclusion in step 2 of the backtrack procedure can be made in a manner consistent with the particular sort being maintained. At any time the sort can be modified in any way that is consistent with the ordering constraints and that reduces the corresponding domain size tuple.

These two heuristics correspond to the two motivations for the partial order backtrack procedure discussed at the beginning of this section. The acceleration heuristic focuses on the topological sort producing the minimal domain size tuple. It is possible to show that under a perfect implementation of the acceleration heuristic the minimal domain size tuple decreases at every backtrack event. However, the topological sort corresponding to the minimal domain size tuple can change between backtracks. This allows the selection of variable ordering to be dynamically improved during the search. The GSAT heuristic corresponds to the second motivation for the partial order backtrack procedure. This heuristic is important in the case where a solution exists and we want to direct the search toward the solution.

To conclude this section we prove that any backtrack event reduces the maximal domain size tuple. We will need the following purely combinatoric lemma.

> **Lemma:** Let Σ be a set of safety conditions containing both $z < w$ and $y < w$. For any topological sort s of $\Sigma - \{y < w\}$ there exists a topological sort s' of Σ such that s' is identical to s up to the variable z.
>
> **Proof:** If w follows y in the sort s then s is a sort of Σ and we are done. If w is before y in s then let s' be the sequence of variables that consists of the sequence of variables prior to w in s followed by by any topological sort of the remaining variables relative to the safety conditions in Σ. Since w must follow z in s, the sort s' is identical to s up to the variable z.

Consider an arbitrary backtrack event that does not generate the empty clause. Such an event consists of two phases. The first phase consists of adding a sequence of nogoods to Γ followed by the execution of the first half of step 5 in which a set of safety conditions of the form $z < w$ are added. The variable z involved in the single execution of step 5 in the backtrack event will be called the *base variable* of the backtrack event. The second phase involves the removal of safety conditions of the form $y < w$ and the removal of irrelevant nogoods as the values of variables are changed. The addition of ordering constraints can only eliminate topological sorts and hence can not increase the maximal domain size tuple. Also note that the live domain size of the base variable z decreases and that every domain size which increases by virtue of the removal of nogoods is constrained (after the backtrack event) to follow z. The removal of safety conditions and nogoods in the second phase of the backtrack event allows for new topological sorts of the variables. It suffices to show that the domain size tuple of any topological sort allowed after the backtrack event is lexicographically smaller than the maximal domain size tuple before the backtrack event. Consider a topological sort s of the order constraints present after the backtrack event. By repeated applications of the above lemma there exists a topological sort s' of the ordering constraints present before the backtrack event such that s' is identical to s up to the base variable z. The domain size tuple associated with s' before the backtrack event can be no larger than the maximal domain size tuple before the backtrack event. Since the domain size of z has been reduced, and all variables whose domain size has increased follow z under the new constraints, the domain size tuple associated with s after the backtrack event must be lexicographically less than the domain size tuple associated with s' before the backtrack event. So the domain size tuple associated with s after the backtrack event must be less than the maximal domain size tuple prior to the backtrack event.

5 Conclusions

Aggressive dependency directed backtracking procedure keep nogoods on future variables and tend to be additive on disjoint unions, i.e., on a union of disjoint subproblems the size of the search space is the sum of the size of the search spaces of the independent subproblems. But of course aggressive dependency directed techniques should also be effective on problems that are not a union of disjoint subproblems. The first polynomial space aggressive dependency directed backtrack procedure seems to be Ginsberg's dynamic backtracking technique.

We have constructed two additional polynomial space aggressive dependency directed backtracking procedures. The first is a generalization of Ginsberg's dynamic backtracking — dynamic backtracking can be simulated by generalized dynamic backtracking under a certain algorithm for selecting variables. Generalized dynamic backtracking allows greater flexibility in dynamic variable ordering — something known to be heuristically important.

The second polynomial space aggressive dependency directed technique uses only a partial order on variables. Intuitively this allows dynamic rearrangement of the order of "past" variables so that "mistakes" in the dynamic selection of variable order can be repaired during the search. The increased flexibility of partial order backtracking can also be motivated by the desire to guide the search toward a solution using "local repairs". The use of a local repair method, also known as the gradient descent method, was one of the original motivations for Ginsberg's dynamic backtracking.

Partial order backtracking is not a generalization of dynamic backtracking. When the live domain of z is reduced dynamic backtracking allows z to be moved to the back of the variable order. More specifically, under dynamic backtracking it is possible that $z < w$ but when the live domain of z is reduced without reducing the live domain of any earlier variables we can get $w < z$. This can not happen under partial order backtracking — under partial order backtracking any variable that must follow the base variable z before the backtrack must still follow the base variable after the backtrack. The fact that partial order backtracking is not a generalization of dynamic backtracking suggests that a yet more flexible algorithm is possible. We leave the construction of such an algorithm as on open problem.

Acknowledgments: I would like to thank Matthew Ginsberg for many useful discussions.

References

[Bruynooghe and Pereira, 1984] M. Bruynooghe and L. M. Pereira. Deduction revision by intelligent backtracking. In J. A. Cambell, editor, *Implementations of Prolog*, pages 196–215. Ellis Horwood, 1984.

[Crawford and Auton, 1993] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of AAAI-93*, pages 21–27, 1993.

[Gaschnig, 1979] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Report, CMU, 1979.

[Ginsberg *et al.*, 1990] M. L. Ginsberg, M. Frank, M. P. Halpin, and M. C. Torrance. Search lessons learned from crossword puzzles. In *Proceedings of AAAI-90*, pages 210–215. Morgan Kaufmann, 1990.

[Ginsberg, 1993] M. Ginsberg. Dynamic backtracking. To appear in the Journal of Artificial Intelligence Research, 1993.

[Minton *et al.*, 1990] S. Minton, M. Johnston, A. Philips, and P. Laird. Solving large scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of AAAI-90*, pages 17–24, 1990.

[Purdom and Robinson, 1981] Cynthia Brown Purdom, Paul Walton and Edward L. Robinson. Backtracking with multi-level dynamic search rearrangement. *Acta Informatica*, 15:99–113, 1981.

[Seidel, 1981] Raimund Seidel. A new method for solving constraint satisfaction problems. In *IJCAI-89*, pages 338–342, August 1981.

[Selman *et al.*, 1992] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of AAAI-92*, pages 440–446, 1992.

[Sosič and Gu, 1991] R. Sosič and J. Gu. Fast search algorithms for the n-queens problem. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1572–1576, 1991.

[Stallman and Sussman, 1977] Richard Stallman and Gerald Sussman. Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[Zabih and McAllester, 1988] Ramin Zabih and David McAllester. A rearrangement search strategy for determining propositional satisfiability. In *AAAI-88*. Morgan Kaufmann Publishers, 1988.