A Theoretical Evaluation of Selected Backtracking Algorithms*

Grzegorz Kondrak and **Peter van Beek**
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1
kondrak@cs.toronto.edu, vanbeek@cs.ualberta.ca

Appears in: *Artificial Intelligence,* 89:365-387, 1997.

Abstract

In recent years, many new backtracking algorithms for solving constraint satisfaction problems have been proposed. The algorithms are usually evaluated by empirical testing. This method, however, has its limitations. Our paper adopts a different, purely theoretical approach, which is based on characterizations of the sets of search tree nodes visited by the backtracking algorithms. A notion of inconsistency between instantiations and variables is introduced, and is shown to be a useful tool for characterizing such well-known concepts as backtrack, backjump, and domain annihilation. The characterizations enable us to: (a) prove the correctness of the algorithms, and (b) partially order the algorithms according to two standard performance measures: the number of nodes visited, and the number of consistency checks performed. Among other results, we prove the correctness of Backjumping and Conflict-Directed Backjumping, and show that Forward Checking never visits more nodes than Backjumping. Our approach leads us also to propose a modification to two hybrid backtracking algorithms, Backmarking with Backjumping (BMJ) and Backmarking with Conflict-Directed Backjumping (BM-CBJ), so that they always perform fewer consistency checks than the original algorithms.

1 Introduction

Constraint-based reasoning is a simple, yet powerful paradigm in which many interesting problems can be formulated. It has received much attention recently, and numerous methods for dealing with constraint networks have been developed. The applications include graph coloring, scene labelling, natural language parsing, and temporal reasoning.

The basic notion of constraint-based reasoning is a constraint network, which is defined by a set of variables, a domain of values for each variable, and a set of constraints between the variables. To solve a constraint network is to find an assignment of values to each variable so that all constraints are satisfied [10, 20].

Backtracking search is one of the methods of solving constraint networks. The generic backtracking algorithm was first described more than a century ago, and since then has been rediscovered many times [2]. In recent years, many new backtracking algorithms have been

*A preliminary version of this paper appeared in *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence,* pages 541-547, Montreal, Quebec, 1995, where it was selected for an Outstanding Paper Award.

proposed. The basic ones include Backmarking [5], Backjumping [6], Forward Checking [8, 11], and Conflict-Directed Backjumping [16]. Several hybrid algorithms, which combine two or more basic algorithms, have also been developed [16].

There is no simple answer to the question of which backtracking algorithm is the best one. First, the performance of backtracking algorithms depends heavily on the problem being solved. Often, it is possible to construct examples of constraint networks on which an apparently very efficient algorithm is outperformed by the most basic chronological backtracking. Second, it is not obvious what measure should be employed for comparison. Run time is not a very reliable measure because it depends on hardware and implementation, and so cannot be easily reproduced. Besides, the cost of performing consistency checks (checks that verify that the current instantiations of two variables satisfy the constraints) cannot be determined in abstraction from a concrete problem. A better measure of the efficiency of a backtracking algorithm seems to be the number of consistency checks performed by the algorithm, although it does not account for the overhead costs of maintaining complex data structures. Another standard measure is the number of nodes in the backtrack tree generated by an algorithm.

The need for ordering algorithms according to their efficiency has been recognized before. Nudel [15] ordered backtracking algorithms according to their average-case performance. Prosser [16] performed a series of experiments to evaluate nine backtracking algorithms against each other. However, such an approach is open to the criticism that the test problems are not representative of the problems that arise in practice. Even a theoretical average-case analysis is possible only if one makes simplifying assumptions about the distribution of problems. Prosser commented on his results:

> It is naive to say that one of the algorithms is the 'champion'. The algorithms have been tested on one problem, the ZEBRA. It might be the case that the relative performance of these algorithms will change when applied to a different problem.

When Prosser's results are examined, it is easy to notice that in some cases one algorithm performed better than another in *all* tested instances. Could this mean that one algorithm is *always* better than another? Such a hypothesis can never be verified solely by experimentation; the relationship has to be proven theoretically. In this paper we show that some of these cases indicate a general rule, whereas other do not. Moreover, we present a partial ordering of several backtracking algorithms which is valid for all instances of all constraint satisfaction problems.

Our approach is purely theoretical. We analyze several backtracking algorithms with the purpose of discovering general rules that determine their behaviour. A notion of inconsistency between instantiations and variables is introduced, and is shown to be a useful tool for characterizing such well-known concepts as backtrack, backjump, and domain annihilation. Using the new notion, we formulate the necessary and sufficient conditions for a search tree node to be visited by each backtracking algorithm. These characterizations enable us to construct partial orders (or *hierarchies*) of the algorithms according to two standard performance measures: the number of visited nodes, and the number of performed consistency checks.

The orderings are surprisingly regular and contain some non-intuitive results. For instance, it turns out that the set of nodes visited by Forward Checking is always a subset of the set of nodes visited by Backjumping. This fact has never been reported before although the two algorithms have been often empirically compared. Also, the orderings confirm and clarify the experimental results published by other researchers. The characterizing conditions imply simple and elegant correctness proofs of the characterized algorithms. Two of these algorithms, Backjumping (BJ) and Conflict-Directed Backjumping (CBJ) have not been formally proven correct before[1].

The orderings proved also to be a stimulus for developing more efficient backtracking algorithms. The idea of combining Backjumping and Backmarking into a new hybrid algorithm was first put forward by Nadel [13]. Such an algorithm, called BMJ, was presented by Prosser [16]. BMJ, however, does not retain all the power of both base algorithms in terms of consistency checks. Prosser observed that on some instances of the zebra problem BMJ performs more consistency checks than BM. In the conclusion of his paper he posed the following question:

> It was predicted that the BM hybrids, BMJ and BM-CBJ, could perform worse than BM because the advantages of backmarking may be lost when jumping back. Experimental evidence supported this. Therefore, a challenge remains. How can the backmarking behaviour be protected?

In this work we answer the question by modifying the two BM hybrids, Backmarking with Backjumping (BMJ), and Backmarking with Conflict-Directed Backjumping (BM-CBJ), so that they always perform fewer consistency checks than both corresponding basic algorithms.

Apart from presenting specific results for particular backtracking algorithms, our goal is also to propose a general methodology: techniques and definitions that can be used for characterizing any backtracking algorithm. This kind of theoretical analysis may be performed for any new backtracking algorithm in order to see if it belongs in the existing hierarchy.

2 Background

We begin with some concepts of the constraint satisfaction paradigm, then give a brief description of four basic backtracking algorithms, and finally present an example that shows the algorithms at work.

Definition 1 *A* binary constraint network [12] *consists of a set of n variables $\{x_1, \ldots, x_n\}$; their respective value domains, $D_1, \ldots, D_n$; and a set of binary constraints. A* binary constraint *or* relation, *R_{ij}, between variables x_i and x_j, is any subset of the product of their domains*[2] *(that is, $R_{ij} \subseteq D_i \times D_j$). We denote an assignment of values to a subset of variables by a tuple of ordered pairs, where each ordered pair (x, a) assigns the value a to the variable x. A tuple is* consistent *if it satisfies all constraints on the variables contained in the tuple. A* (full) solution *of the network is a consistent tuple containing all variables. A* partial solution *of the network is a consistent tuple containing some variables. For simplicity, we usually abbreviate $((x_1, a_1), \ldots, (x_i, a_i))$ to $(a_1, \ldots, a_i)$.*

The next definition introduces a notion of consistency between a tuple of instantiations and a set of variables. This notion is fundamental to all results presented in this work.

Definition 2 *A tuple $((x_{i_1}, a_{i_1}), \ldots, (x_{i_u}, a_{i_u}))$ is* consistent with a set of variables *$\{x_{j_1}, \ldots, x_{j_v}\}$ if there exist instantiations $a_{j_1}, \ldots, a_{j_v}$ of the variables $x_{j_1}, \ldots, x_{j_v}$ respectively, such that the tuple $((x_{i_1}, a_{i_1}), \ldots, (x_{i_u}, a_{i_u}), (x_{j_1}, a_{j_1}), \ldots, (x_{j_v}, a_{j_v}))$ is consistent*[3]*. A tuple is* consistent with a variable *if it is consistent with a one-element set containing this variable.*

[1] Both BJ and CBJ were first presented without correctness proofs and no direct proofs of these algorithms have appeared in the literature. However, proofs have been given for certain algorithms related to CBJ [3, 7, 18].

[2] Throughout the paper we assume that all domain values satisfy the corresponding unary constraints.

[3] The variables in the tuple and in the set of variables need not be distinct. We assume, however, that a variable is always assigned only a unique value.

Example 1. The n-queens problem is how to place n queens on a $n \times n$ chess board so that no two queens attack each other. There are several possible representations of this problem as a constraint network (see [14]). The one we use identifies board columns with variables, and rows with domain values. Thus, variable x_i represents the i-th column, and its domain D_i contains n values representing each row. The constraint between variables x_i and x_j can be expressed as $R_{ij} = \{(a_i, a_j) : (a_i \neq a_j) \wedge (|i - j| \neq |a_i - a_j|)\}$. Figure 1 shows two instances of the 4-queens problem. The instance on the left depicts tuple $((x_1, 4), (x_2, 2))$, which is a partial solution. The tuple is itself consistent and it is consistent with the set of variables $\{x_1, x_2, x_4\}$ and all its subsets, including the empty set. It is inconsistent with all sets of variables that include x_3. It is consistent with variables x_1, x_2, and x_4, but not with variable x_3. The instance on the right depicts tuple $((x_1, 2), (x_2, 4), (x_3, 1), (x_4, 3))$, or simply (2,4,1,3), which is a full solution. The tuple is consistent with all sets of variables. Since the network has a solution, the empty tuple is also consistent with all sets of variables.

Figure 1: A partial and a full solution to the 4-queens problem. The shaded squares denote the positions which are excluded from consideration by the already placed queens.

The idea of a *backtracking algorithm* is to extend partial solutions. At every stage of backtracking search, there is some *current partial solution* which the algorithm attempts to extend to a full solution. Each variable occurring in the current partial solution is said to be *instantiated* to some value from its domain. For ease of exposition, we assume the static *order of instantiation* in which variables are added to the current partial solution according to the predefined order: $x_1, \ldots, x_n$. (This assumption is later relaxed in Section 6.) It is convenient to divide all variables into three sets: *past variables* (already instantiated), *current variable* (now being instantiated), and *future variables* (not yet instantiated). A *dead-end* occurs when all values of the current variable are rejected by a backtracking algorithm when it tries to extend a partial solution. In such a case, some instantiated variables become *uninstantiated*; that is, they are removed from the current partial solution. This process is called *backtracking*. If only the most recently instantiated variable becomes uninstantiated then it is *chronological backtracking*. Otherwise, it is *backjumping*. A backtracking algorithm terminates when all possible assignments have been tested or a certain number of solutions have been found.

A backtrack search may be seen as a *search tree* traversal. In this approach we identify tuples (assignments of values to variables) with nodes: the empty tuple is the root of the tree, the first level nodes are 1-tuples (representing an assignment of a value to variable x_1), the second level nodes are 2-tuples, and so on. The levels closer to the root are called shallower levels, and the levels farther from the root are called deeper levels. Similarly, the variables corresponding to these levels are called shallower and deeper. The nodes that represent consistent tuples are called *consistent nodes*. The nodes that represent inconsistent tuples are called *inconsistent nodes*. We say that a backtracking algorithm *visits* a node if at some stage of the algorithm's execution the instantiation of the current variable and the instantiations of the past variables form the tuple identified with this node. The nodes visited by a backtracking algorithm form a subset of the set of all nodes belonging to the search tree. We call this subset, together with the connecting edges, the *backtrack tree* generated by a backtracking algorithm. Backtracking itself can be seen as retreating to shallower levels of the search tree. Whenever some variables become uninstantiated and x_h is set as the new current variable, we say that the algorithm backtracks to level h. We consider two backtracking algorithms to be equivalent if on every constraint network they generate the same backtrack tree and perform the same consistency checks.

Chronological Backtracking (BT) [2] is the generic backtracking algorithm. The consistency checks between the instantiation of the current variable and the instantiations of the past variables are performed according to the original order of instantiations. If a consistency check fails, the next domain value of the current variable is tried. If there are no more domain values left, BT backtracks to the most recently instantiated past variable. If all checks succeed, the branch is extended by instantiating the next variable to each of the values in its domain. A solution is recorded every time that all consistency checks succeed after the last variable has been instantiated.

Backjumping (BJ) [6] is similar to BT, except that it behaves more efficiently when no consistent instantiation can be found for the current variable x_i (at a dead-end). Instead of chronologically backtracking to the preceding variable, BJ backjumps to the deepest past variable x_h that was checked against the current variable. Changing the instantiation of x_h may allow a consistent instantiation to be found for x_i, whereas changing the instantiation of any of the variables between x_i and x_h is guaranteed to be fruitless since we will not have changed the reason for the dead-end.

Conflict-Directed Backjumping (CBJ) [16] has a more sophisticated backjumping behaviour than BJ. Every variable has its own *conflict set* that contains the past variables which failed consistency checks with its current instantiation. Every time a consistency check fails between an instantiation a_i of the current variable x_i and an instantiation a_h of some past variable x_h, the variable x_h is added to the conflict set of x_i. When there are no more values to be tried for the current variable x_i, CBJ backtracks to the deepest variable x_h in the conflict set of x_i. At the same time, the variables in the conflict set of x_i, with the exception of x_h, are added to the conflict set of x_h, so that no information about conflicts is lost.

In contrast with the above *backward checking* algorithms, Forward Checking (FC) [8, 11] performs consistency checks *forward*, that is, between the current variable and the future variables. After the current variable has been instantiated, the domains of the future variables are filtered in such a way that all values inconsistent with the current instantiation are removed. If none of the future domains is annihilated, the next variable becomes instantiated to each of the values in its filtered domain. Otherwise the effects of forward checking are undone, and the next value is tried. If there are no more values to be tried for the current variable, FC backtracks chronologically to the most recently instantiated variable. A solution is recorded every time the last variable becomes instantiated.

Example 2. Figure 2 shows a fragment of the backtrack tree generated by Chronological Backtracking (BT) for the 6-queens problem. White dots denote consistent nodes. Black dots denote inconsistent nodes. For simplicity, when referring to nodes we omit commas and parentheses. The board in the upper right corner depicts the placing of queens corresponding to node 253 in the backtrack tree. Capital Q's on the board represent queens which have already been placed on the board. The shaded squares represent positions that must be excluded due to the already placed queens. The numbers inside the squares indicate the first queen responsible for the exclusion; 1,2,3 correspond to the first, second, and third queen respectively.

The dark-shaded part of the tree contains two nodes that are skipped by Backjumping (BJ). The algorithm detects a dead-end at variable x_6 when it tries to expand node 25364. It then backjumps to the deepest variable in conflict with x_6, in this case x_4. The backjump is represented by a dashed arrow. We could say that BJ discovers that the tuple (2,5,3,6),

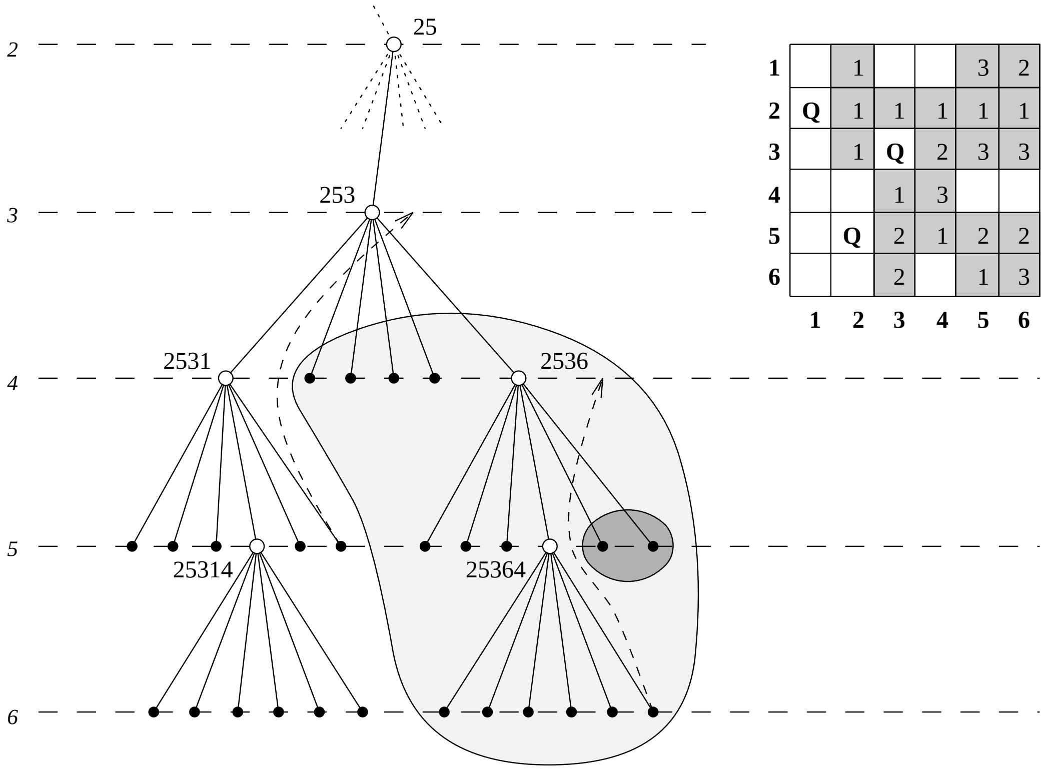

Figure 2: A fragment of the BT backtrack tree for the 6-queens problem.

which is composed of the instantiations in conflict with x_6, is inconsistent with variable x_6. To see this, notice that if we place a queen in column 4 row 6, every square in column 6 is attacked by the queens placed in the first four columns. Indeed, there is no point in trying out the remaining values for x_5 because that variable plays no role in the detected inconsistency. Nodes 25365 and 25366 may be safely skipped.

The light-shaded part of the tree contains nodes that are skipped by Conflict-Directed Backjumping (CBJ). The algorithm reaches a dead-end when expanding node 25314. At this moment the conflict set of x_6 is $\{1, 2, 3, 5\}$ because the instantiations of these four variables prevent a consistent instantiation of variable x_6. To see this, notice that after the fourth and the fifth queen are placed, column 6 of the chess board will contain numbers $1, 2, 3$, and 5. CBJ backtracks to the deepest variable in the conflict set, which is x_5. No nodes are skipped at this point. The conflict set of x_6 is added to the conflict set of x_5, which now becomes $\{1, 2, 3\}$. After trying the two remaining values for x_5, CBJ backjumps to x_3 skipping the rest of the subtree. The backjump is represented by a dashed arrow. In terms of consistency, we could say that the algorithm discovered that tuple (2,5,3) is inconsistent with the set of variables $\{x_5, x_6\}$. A look at the board in Figure 2 convinces us that indeed such a placement of queens cannot be extended to a full solution. It is impossible to fill columns 5 and 6 simply because the two available squares are in the same row. Note that (2,5,3) is consistent with both x_5 and x_6 taken separately.

Forward Checking (FC), in contrast with the backward checking algorithms, visits only consistent nodes, although not necessarily all of them. In our example, nodes 253, 2531, 25314 and 2536 are visited, but not 25364. The board in Figure 2 can be interpreted in the context of this algorithm as follows. The shaded numbered squares correspond to the values filtered from domains of variables by forward checking. The squares that are left empty as the search progresses correspond to the nodes visited by FC. Due to the filtering scheme, FC detects an inconsistency between the current partial solution and some future variable without ever reaching that variable, but it is unable to discover an inconsistency with a *set* of variables. In our example, the algorithm finds that both 25314 and 2536 are inconsistent with x_6. However, it does not discover that node 253 is inconsistent with $\{x_5, x_6\}$. That is why node 2536 is visited by FC even though it is skipped by the backward checking CBJ.

3 Characterizations of Four Basic Algorithms and Their Implications

We are now ready to present some new results. First, we give two lemmas that define backjumps in terms of inconsistency between variables and instantiations. Then, we present theorems about the backtrack trees of the four basic backtracking algorithms: BT, BJ, CBJ, and FC. The theorems enable us to (a) partially order the algorithms according to the number of visited nodes, and (b) prove the correctness of the algorithms. It is assumed that all constraints are binary, the order of instantiations is fixed and static, and the order of performing consistency checks within the node follows the order of instantiations. When faced with a constraint satisfaction problem one can ask several questions about it [15]: Is there a solution? How many solutions are there? What is one solution? What are all the solutions? We focus first on those variants of the backtracking algorithms that find all solutions. We make the assumption of a static variable ordering and the assumption that all solutions are sought in order to simplify the statements of the results and their proofs. These two assumptions are later relaxed in Section 6. The proofs that are not included here can be found in [9].

In Example 2 we made an observation concerning the relation between a BJ backjump and the consistency of the current instantiation. Let us generalize this observation in the form of the following lemma.

Lemma 1 *If BJ performs a backtrack to variable x_h from a dead-end at variable x_i then $(a_1, \ldots, a_h)$ is inconsistent with x_i.*

Proof. After no consistent instantiation can be found for x_i, BJ chooses as the point of backtrack the variable x_h which is the deepest variable in conflict with x_i. Let C denote the tuple composed of the instantiations of all variables that are in conflict with x_i. Clearly, C is inconsistent with x_i. Since a_h is the instantiation of the deepest variable in C, C is a subtuple of $(a_1, \ldots, a_h)$. Therefore, $(a_1, \ldots, a_h)$ is also inconsistent with x_i. $\square$

In order to present a similar lemma for the CBJ algorithm, we need to consider two additional issues. The first issue concerns the one solution/all solutions dichotomy. Backtracking algorithms are usually designed to stop after finding the first solution and have to be modified in order to find all solutions. For many algorithms, including BT, BJ, and FC, the changing of the termination condition is sufficient. In the case of CBJ and its hybrids, however, a more substantial modification is necessary. Recall that the conflict sets of CBJ are meant to indicate which instantiations are responsible for a previously discovered inconsistency. However, after a solution is found, conflict sets cannot be interpreted in this way. It is the search for other solutions, rather than an inconsistency, that forces the algorithm to backtrack. We need to differentiate between these two types of CBJ backtracks, namely (A-type) the backtracks caused by detecting an inconsistency, and (B-type) the backtracks caused by searching for other solutions. In the latter case the backtrack must be always chronological (i.e., to the immediately preceding variable) and no nodes can be skipped, otherwise we would risk pruning out solutions. One possible solution is to add to every conflict set a flag that indicates whether the conflict set is valid. If the *vcf (valid conflict* *set)* flag is set, the deepest variable in the conflict set should be taken as the backtrack point; otherwise, a chronological B-type backtrack must be applied. When a solution is found, all *vcf* flags should be cleared.

The second issue concerns the ability of CBJ to perform multiple backjumps. To deal with this problem, we need the notion of *backtrack rank* for the A-type backtracks. Informally, the rank of a backtrack is the distance, measured in backtracks, from the backtrack destination to the "farthest" dead-end. The definition is recursive:

Definition 3

1. *A backtrack from variable x_i to variable x_h is of rank 1 if it is performed directly from a dead-end at x_i.*

2. *A backtrack from variable x_i to variable x_h is of rank $d \geq 2$, if all backtracks performed to variable x_i are of rank less than d, and at least one of them is of rank $d-1$.*

The following lemma describes the relation between a CBJ backjump and the consistency of the current instantiation.

Lemma 2 *If CBJ performs an A-type backtrack from variable x_i to variable x_h, then there exists a set of variables S such that S is a subset of $\{x_i, \ldots, x_n\}$ containing x_i and the tuple composed of the instantiations of the variables in the conflict set of x_i is inconsistent with S.*

Proof. Recall that CBJ chooses as the point of backtrack the deepest variable in the conflict set of the current variable. The conflict set of x_i is the union of the set of all past variables in conflict with x_i and all conflict sets inherited from variables deeper than x_i. Let C denote the tuple composed of the instantiations of the variables in the conflict set of x_i.

The proof proceeds by induction on the rank of the backtrack. For the basis, consider a backtrack of rank 1, that is, one performed from a dead-end. Since no conflict sets are inherited from deeper variables, the conflict set of x_i contains only variables in conflict with x_i. Clearly, C is inconsistent with the set $S = \{x_i\}$. (Note that in this case the behaviour of CBJ is identical to that of BJ.)

Now, assume the inductive hypothesis is true for all backtracks of rank less than d and consider a backtrack of rank d. We want to find a set S such that C is inconsistent with it. Let $C^{(x_i,t)}$ denote the tuple produced by extending C with some instantiation $(x_i,t), t \in D_i$. $C^{(x_i,t)}$ itself may be consistent or not.

A) If $C^{(x_i,t)}$ is a consistent tuple, there must have been a backtrack of rank less than d from some variable x^t to variable x_i. From the inductive hypothesis we know that the tuple C^t composed of the instantiations of the variables in the conflict set of x^t is inconsistent with some set S^t. Since the conflict set of x_i contains all elements of the conflict set of x^t except x_i, C^t is a subtuple of $C^{(x_i,t)}$, and so the latter is also inconsistent with S^t.

B) If $C^{(x_i,t)}$ is an inconsistent tuple, it is also inconsistent with any set of variables, so take $S^t = \emptyset$.

Let S' be the sum of all the S^t sets, $S' = \bigcup_{t \in D_i} S^t$. For every instantiation $(x_i,u), u \in D_i$, $C^{(x_i,u)}$ is inconsistent with S'. Therefore C is inconsistent with the set $S = \{x_i\} \cup S'$. □

We now present two theorems that specify the sufficient and the necessary conditions respectively, for a node to be visited by the four basic backtracking algorithms. The first theorem can be interpreted as a description of the sets of nodes which are guaranteed to be visited by the algorithms. The assumption is that all solutions are sought.

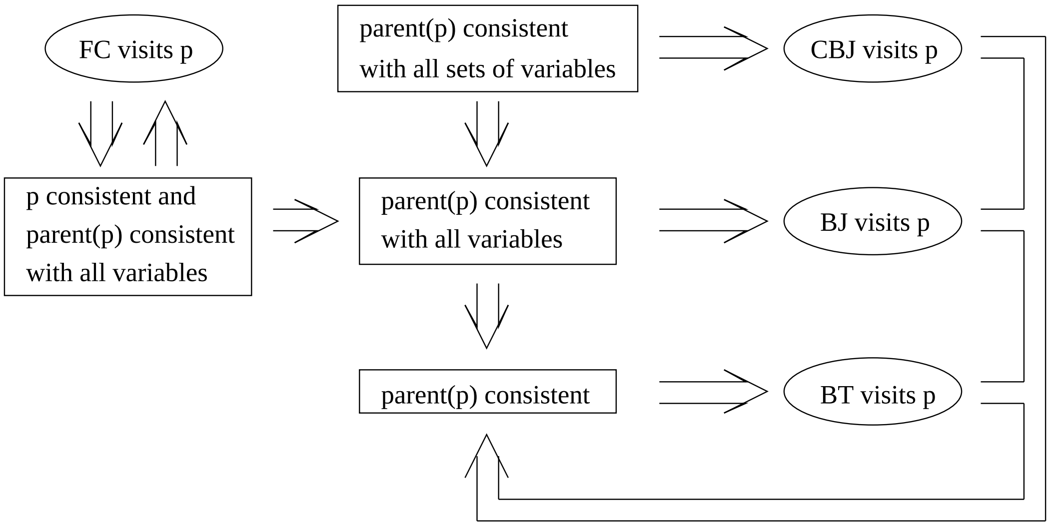

Figure 3: Conditions graph.

Theorem 1

a) *If the parent of a node is consistent, then BT visits the node.*

b) *If the parent of a node is consistent with every variable, then BJ visits the node.*

c) *If the parent of a node is consistent with every set of variables, then CBJ visits the node.*

d) *If a node is consistent and its parent is consistent with every variable, then FC visits the node.*

Proof.

b) Suppose that node $(a_1, \ldots, a_{i-1})$ is consistent with every variable, and its child $p = (a_1, \ldots, a_i)$ is not visited by BJ. Take the deepest j such that node $p' = (a_1, \ldots, a_j)$ is visited by BJ. Node p' is a proper ancestor of node p and is consistent with every variable. When BJ is at node p', all consistency checks between a_j and previous instantiations succeed. The only reason for not instantiating the next variable x_{j+1} to a_{j+1} can be a backjump from some variable x_h to some variable x_g, where $g \leq j$ and $h \geq j + 2$. But if this is the case, Lemma 1 implies that node $(a_1, \ldots, a_g)$ is inconsistent with x_h, which contradicts the initial assumption that node $(a_1, \ldots, a_{i-1})$ is consistent with every variable.

c) Similar to the proof of b), except that we use Lemma 2. Note that we are concerned here only with the A-type backtracks because the B-type backtracks are always chronological and do not involve node skipping.

Proofs of the remaining cases are straightforward. □

The next theorem can be seen as describing the sets of nodes that *may* be visited by the algorithms, or, if we consider their complements, the sets of nodes that are never visited by the algorithms.

Theorem 2

a) If BT visits a node, then its parent is consistent.

b) If BJ visits a node, then its parent is consistent.

c) If CBJ visits a node, then its parent is consistent.

d) If FC visits a node, then it is consistent and its parent is consistent with every variable.

Proof.

a)-c) The proofs follow from the fact that the backward checking algorithms expand only consistent nodes.

d) We prove the second conjunct first. Suppose that FC visits node $p = (a_1, \ldots, a_i)$ although its parent $(a_1, \ldots, a_{i-1})$ is inconsistent with some variable. Take the deepest j, $j < i$, such that node $(a_1, \ldots, a_{j-1})$ is consistent with every variable. Node $p' = (a_1, \ldots, a_j)$ is a proper ancestor of node p, so p' is also visited by FC. When FC is at node p', consistency checking annihilates the domain of some variable, thus causing the branch to be abandoned. Therefore, no descendants of p' are visited by FC, a contradiction.

Now, suppose that FC visits node $p = (a_1, \ldots, a_i)$ which is inconsistent. From the first part of the proof, we know that its parent $(a_1, \ldots, a_{i-1})$ must be consistent. Take the shallowest k, $k < i$, such that instantiation a_k is inconsistent with instantiation a_i. When FC is at node $(a_1, \ldots, a_k)$, the value a_i is removed from the domain of the variable x_i and cannot be reinstated before the instantiation of x_k is changed. Therefore, p cannot be visited by FC, a contradiction. □

Figure 3 summarizes the results presented so far. The arrows represent implications formulated in Theorems 1 and 2. Note the difference between the chronologically backtracking algorithms BT and FC, and the backjumping algorithms BJ and CBJ. The former are completely characterized as the necessary and sufficient conditions coincide; for every node we can decide whether it is visited by the algorithm without generating the whole backtrack tree. The latter are only partially characterized; there is a set of nodes for which we are unable to tell *a priori* if they belong to the algorithm's search tree or not. It is an open question if better characterizing conditions for the backjumping algorithms can be found.

The following corollary has been formulated by simply following the arrows in Figure 3.

Corollary 1

a) BT visits all nodes that BJ visits.

b) BT visits all nodes that CBJ visits.

c) BT visits all nodes that FC visits.

d) BJ visits all nodes that FC visits.

The relationship between BJ and FC is the most interesting. It has never been reported before, although the two algorithms have been often empirically compared.

A relationship between BJ and CBJ, although not implied by the theorems, can also be proven using Lemmas 1 and 2. These relationships and more are summarized in Figure 7.

Theorem 3 *BJ visits all nodes that CBJ visits.*

Proof. Suppose that in the search tree of CBJ there is a node $p = (p_1, \ldots, p_h)$ which is not visited by BJ (Figure 4, left). The only reason for skipping p can be a backjump performed by BJ from some node $q = (q_1, \ldots, q_k)$ to level $g < h$. Recall that BJ performs backjumps only immediately after detecting a dead-end, and that in such a case it behaves exactly like CBJ. Therefore, node q cannot be visited by CBJ, otherwise CBJ would also skip node p. The only reason for skipping q can be a backjump performed by CBJ from some node $r = (r_1, \ldots, r_j)$ to level $i < k$ (Figure 4, right).

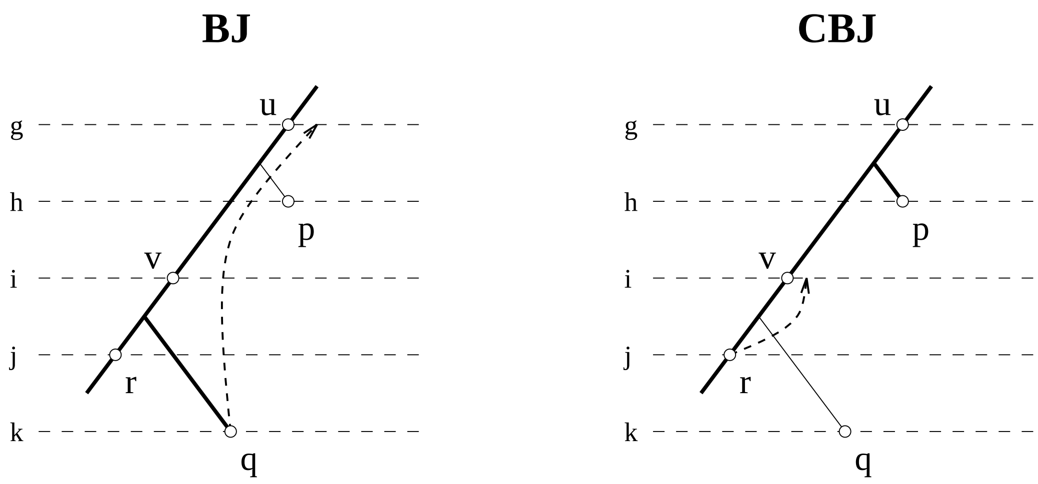

Figure 4: A hypothetical situation where CBJ visits a node not visited by BJ.

Let $u = (p_1, \ldots, p_g) = (q_1, \ldots, q_g) = (r_1, \ldots, r_g),$, and $v = (q_1, \ldots, q_i) = (r_1, \ldots, r_i)$. From Lemma 1 we have that u is inconsistent with variable x_k. From Lemma 2 we have that v is inconsistent with set S, where $S \subseteq \{x_j, \ldots, x_n\}$.

Let us denote the deepest variable in S by $max(S)$. What is the relationship between x_k and $max(S)$?

- If $x_k > max(S)$, BJ would never reach x_k after visiting node v because it would hit a dead-end at $max(S)$ first.
- If $x_k < max(S)$, CBJ would never reach $max(S)$ after visiting node u because it would hit a dead-end at x_k first.
- If $x_k = max(S)$, CBJ would not visit node p because from x_k it would jump back directly to level g.

Thus, we arrive at a contradiction. □

Corollary 1 together with Theorem 3 enable us to construct a partial order of backtracking algorithms with respect to the number of visited nodes. BT generates the largest backtrack tree, which contains all nodes visited by the other algorithms. BJ visits more nodes than CBJ or FC. The order would be linear if there was a relationship between FC and CBJ, but this is not the case. Figure 2 provides a counterexample: some nodes visited by CBJ are not visited by FC, and vice versa.

The correctness of the four basic algorithms is also an almost immediate consequence of the theorems. A backtracking algorithm is correct if it is sound (finds only solutions), complete (finds all solutions), and terminates. That all the algorithms terminate is clear, so only soundness and completeness have to be shown.

Corollary 2

a) *BT is correct.*

b) *BJ is correct.*

c) *CBJ is correct.*

d) *FC is correct.*

Proof.

b) *Soundness.* A solution is claimed by BJ if all consistency checks succeed at an n-level node. This means that $(a_1, \ldots, a_n)$ is visited and $\forall i < n : a_i$ is consistent with a_n. Theorem 2 implies that node $(a_1, \ldots, a_{n-1})$ is consistent. Therefore, $(a_1, \ldots, a_n)$ is consistent.

Completeness. Suppose that some n-level node $(a_1, \ldots, a_n)$ in the search tree is consistent. Then, its parent $(a_1, \ldots, a_{n-1})$ is consistent as well, and it is also consistent with x_n. Therefore, $(a_1, \ldots, a_{n-1})$ is consistent with every variable. From Theorem 1 we know that $(a_1, \ldots, a_n)$ is visited by BJ. Since all consistency checks between a_n and previous instantiations must succeed, a solution is claimed by BJ.

Proofs of the remaining cases are similar. $\Box$

4 Backmarking and its Hybrids

In this section we discuss Backmarking (BM) [5] and its two hybrids. We prove the correctness of BM and propose a modification to the hybrid algorithms. These algorithms are then included in our hierarchies (see Section 6).

In the Chronological Backtracking (BT) algorithm consistency checks are performed unconditionally. A consistency check is performed to determine if the current instantiations of two variables satisfy the constraint between the variables even if neither of the instantiations has changed since the check was most recently executed. Backmarking (BM) [5] addresses this inefficiency by imposing a *marking scheme* on the Chronological Backtracking algorithm. The marking scheme employed by BM and its hybrids does not have any influence on the backtrack tree generated by a backtracking algorithm but usually results in a dramatic reduction in the number of consistency checks. It is based on the following two observations [13]:

A) If, at the most recent node where a given instantiation was checked, the instantiation failed against some past instantiation that has not yet changed, then it will fail against it again. Therefore, all consistency checks involving it may be avoided.

B) If, at the most recent node where a given instantiation was checked, the instantiation succeeded against all past instantiations that have not yet changed, then it will succeed against them again. Therefore we need to check the instantiation only against the more recent past instantiations which have changed.

The above two statements can be formally proven correct using our framework.

Lemma 3 *The marking scheme formulated by the observations A and B is correct.*

Proof. Let $p = (a_1, \ldots, a_i)$ be a node visited by a backward checking algorithm. Node p may be consistent or not. If p is a consistent node then

$$\forall j < i : \ (a_j, a_i) \in R_{ji}.$$

If p is an inconsistent node then

$$\exists!\ s < i : (((a_s, a_i) \notin R_{si}) \wedge (\forall j < s :\ (a_j, a_i) \in R_{ji})),$$

where $\exists!$ means *there uniquely exists*. Let $p' = (a'_1, \ldots, a'_i)$ be the first node visited after p such that $a'_i = a_i$. We have

$$\exists!\ r < i : ((a'_r \neq a_r) \wedge (\forall j < r : a'_j = a_j)).$$

There are now two possibilities, which correspond exactly to the observations A and B:

A) $[\neg consistent(p) \wedge (s < r)] \ \Rightarrow\ [(a'_s, a'_i) \notin R_{si}] \ \Rightarrow\ [\neg consistent(p')]$

B) $[consistent(p) \vee (\neg consistent(p) \wedge (s \geq r))] \ \Rightarrow\ [\forall j < r :\ (a'_j, a'_i) \in R_{ji}]$

□

BM is essentially BT enhanced by the above marking scheme. Its standard implementation uses a one-dimensional array *mbl* (minimum backup level) of size n and a two-dimensional array *mcl* (maximum checking level) of size $n \times m$, where n is the number of variables, and m is the size of the largest domain. The entry *mbl*[i] contains the number of the shallowest variable whose instantiation has changed since the variable x_i was last instantiated with a new value. The entry *mcl*[i][j] contains the number of the deepest variable that was checked against the j-th value in the domain of the variable x_i. All entries in both arrays are initially set to 1. Roughly speaking, *mbl* holds the values of r, and *mcl* holds the values of s. For the implementation details see for example [16].

From a theoretical point of view, BM may be treated as an abstract algorithm which has a number of possible implementations. Within this approach, proving the correctness of the marking scheme is in fact equivalent to proving the correctness of BM.

Theorem 4 *BM is correct.*

Proof. Since BM is BT enhanced by the marking scheme formulated by the observations A and B, the correctness of BT and the correctness of the marking scheme imply the correctness of BM. □

BM generates exactly the same search tree as BT, but often performs less checks within a node. This is in contrast with BJ, which reduces the number of consistency checks by skipping search tree nodes. It turns out that the two types of savings can be incorporated into one backtracking algorithm. Nadel [13] was the first to suggest combining BM and BJ into a new hybrid algorithm. Prosser [16] presented such an algorithm, called Backmarking and Backjumping (BMJ). BMJ, however, does not retain all the power of each base algorithm in terms of consistency checks. Prosser observed that on some instances of the zebra problem BMJ performs more consistency checks than BM. BMJ is also worse than BM on the benchmark 8-queens problem.

Example 3. Consider the constraint network of four variables represented by the graph in Figure 5. The domains of the variables are given inside the nodes, and the constraints between variables are specified by the allowed pairs along the arrows. The search is performed in the order x_1, x_2, x_3, x_4. It is easy to verify that there is only one solution to the network. Figure 6 shows the backtrack tree generated by BT, which performs 17 consistency checks on this constraint network. In comparison with BT, BM saves one consistency check on each of the nodes numbered 8 to 11, which brings down the total number of consistency checks to 13. The saving on node 8 corresponds to observation A in the marking scheme, while the other three savings correspond to observation B. BMJ saves two consistency checks by backjumping over node 6 but on the whole performs 14 checks.

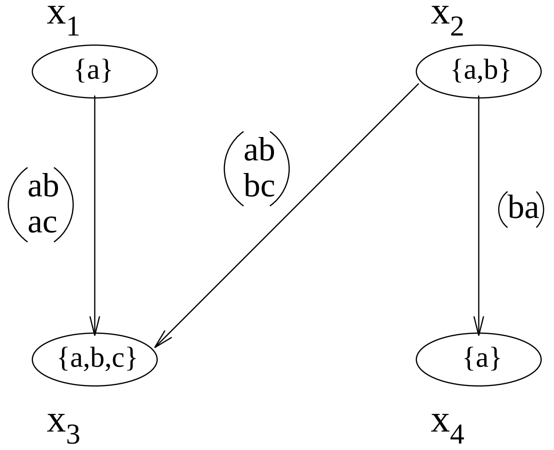

Figure 5: The constraint network of Example 3.

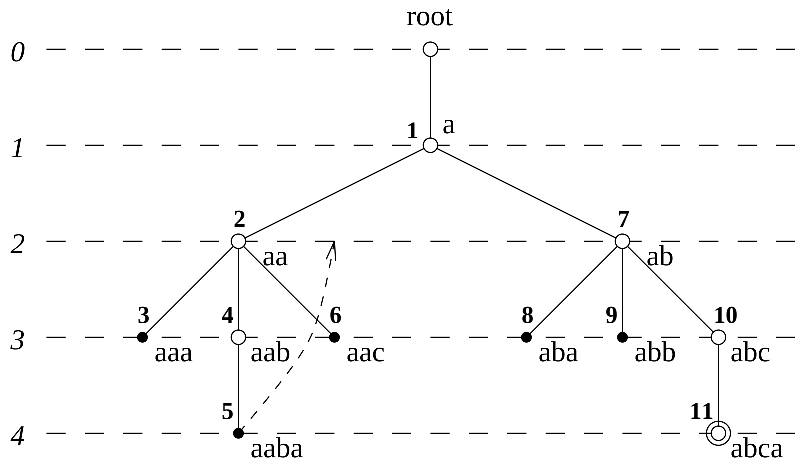

Figure 6: The backtrack tree generated by BT on the constraint network of Example 3.

A careful analysis of the above example leads us to the conclusion that BMJ is sometimes worse than BM because it does not implement the above marking scheme accurately. The one-dimensional *mbl* array, which was originally designed for a chronologically backtracking algorithm, is no longer adequate for a backjumping algorithm. BM always instantiates a variable in turn to all possible values in its domain. Therefore, the r-values are the same throughout the domain of a variable, and a single *mbl* entry is sufficient to hold them all. In BMJ, however, because of backjumps, not all values in the domain are always tested. When this happens, the r-values may differ within the domain. The loss of information caused by the inadequacy of the *mbl* array is the sole reason why BMJ is sometimes outperformed by BM. In such cases, the number of redundant checks performed by BMJ exceeds the number of checks avoided by the node skipping.

We propose a modified BackMarkJump (BMJ2), which solves the problem by making *mbl* a two-dimensional rather than a one-dimensional array. The new *mbl* array is of size $n \times m$, so that each *mcl* entry has a corresponding *mbl* entry (this is a reasonable space requirement because BMJ already uses one $n \times m$ array). Each *mcl* entry now has a corresponding *mbl* entry. A separate entry for each domain value makes it possible to preserve all collected consistency information. The $mbl[i][j]$ entry stores the number of the shallowest variable whose instantiation has changed since the variable x_i was last instantiated with the j–th value. As in the case of BM, the correctness of BMJ2 is a consequence of the correctness of the marking scheme and the correctness of the underlying algorithm (BJ).

BMJ2 is not only never worse than BMJ, but also never worse than BM. The set of nodes visited by BMJ2 is the same as the set of nodes visited by BJ and BMJ, and is a subset of the nodes visited by BM. At any given node BMJ2 performs no more consistency checks than BJ or BMJ. It uses the same marking scheme as BM and therefore is never worse than BM. However, thanks to its backjumping abilities, BMJ2 makes additional savings by skipping nodes, which explains why it often performs less consistency checks than BM. On the constraint network of Example 3 (see Figures 5 and 6), BMJ2 performs only 12 consistency checks.

An analogous modification of Backmarking and Conflict-Directed Backjumping (BM-CBJ), which is another hybrid proposed by Prosser, produces BM-CBJ2: *mbl* should be made a 2-dimensional array, and maintained in the same way as in BMJ2.

5 The Hybrid Algorithm FC-CBJ

In this section we discuss the hybrid algorithm Forward Checking and Conflict-Directed Backjumping (FC-CBJ) [16]. We prove the correctness of the algorithm and characterize the set of search tree nodes visited by the algorithm. The algorithm is then included in our hierarchies (see Section 6).

FC-CBJ, proposed by Prosser [16], is an attempt to combine the advantages of FC and CBJ. In contrast with FC, which always backtracks chronologically, FC-CBJ records the information about the variables that caused current inconsistency, and later uses this information to determine the backtracking point. Every time a consistency check fails between the instantiation a_i of the current variable x_i and an instantiation of some future variable x_j, the variable x_i is added to the conflict set of x_j. Every time a domain annihilation of a variable x_k occurs, the variables in the conflict set of x_k are added to the conflict set of the current variable x_i. When there are no more values to be tried for the current variable x_i, FC-CBJ backtracks to the deepest variable x_h in the conflict set of x_i. At the same time, the variables in the conflict set of x_i, with the exception of x_h, are added to the conflict set of x_h, so that no information about conflicts is lost.

FC-CBJ was identified by Prosser as the champion among the nine backtracking algorithms that he tested on the zebra problem. More recently B. Smith [19] observed that a variant of FC-CBJ performs well on exceptionally hard problems. It is therefore important to characterize and prove the correctness of this algorithm.

Let us start by determining the necessary and sufficient conditions for a search tree node to be visited by FC-CBJ. The necessary condition for FC-CBJ is the same as for FC.

Theorem 5 *If FC-CBJ visits a node, then it is consistent and its parent is consistent with every variable.*

Proof. Similar to the proof of Theorem 2, case (d). □

The above theorem together with Theorem 1 imply that if a node is visited by FC-CBJ, it is also visited by FC. In the worst case, FC-CBJ visits the same set of nodes as FC. However, since there exist constraint networks on which FC-CBJ visits less nodes than FC, we can place FC-CBJ in the node hierarchy directly below FC (see Figure 7). The relationship holds also for the checks hierarchy because at any given node FC-CBJ performs exactly the same number of consistency checks as FC (see Figure 8).

In order to obtain the sufficient condition for FC-CBJ, it is necessary to formulate an equivalent of Lemma 2. Surprisingly, the following lemma is virtually identical to Lemma 2. The statement of the lemma uses the concept of A-type backtracks defined in Section 3.

Lemma 4 *If FC-CBJ performs an A-type backtrack from variable x_i to variable x_h, then there exists a set of variables S such that S is a subset of $\{x_i, \ldots, x_n\}$ containing x_i and the tuple composed of the instantiations of the variables in the conflict set of x_i is inconsistent with S.*

Proof. The proof is similar to the proof of Lemma 2. First, observe that when a domain annihilation of variable x_k occurs, we have the case of inconsistency of the current tuple $(a_1, \ldots, a_i)$ with the variable x_k. In a forward checking algorithm a dead-end occurs when every instantiation of the current variable either has already been filtered or causes annihilation of the domain of some future variable. The above definition of a dead-end allows us to adopt here without any change the definition of backtrack rank from Section 3 for the CBJ algorithm.

The proof proceeds by induction on the rank of the backtrack. For the basis, consider a backtrack of rank 1, that is, one performed from a dead-end. Let C denote the tuple composed of the instantiations of the variables in the conflict set of x_i. We want to find a set S such that C is inconsistent with it. Let $C^{(x_i,t)}$ denote the tuple produced by extending C with some instantiation $(x_i, t), t \in D_i$. $C^{(x_i,t)}$ itself may be consistent or not.

A) Assume that $C^{(x_i,t)}$ is a consistent tuple. Since all variables that filter values from the domain of x_i are included in the conflict set of x_i, and $C^{(x_i,t)}$ is consistent, t could not have been filtered from the domain of x_i. Furthermore, because it is a dead-end, domain annihilation of some variable x^t must have occurred. Therefore, $C^{(x_i,t)}$ is inconsistent with the one-element set $S^t = \{x^t\}$.

B) If $C^{(x_i,t)}$ is an inconsistent tuple, it is also inconsistent with any set of variables, so take $S^t = \emptyset$.

Let S' be the sum of all the S^t sets, $S' = \bigcup_{t \in D_i} S^t$. For every instantiation $(x_i, u), u \in D_i$, $C^{(x_i,u)}$ is inconsistent with S'. Therefore C is inconsistent with the set $S = \{x_i\} \cup S'$.

The remaining part of the proof is identical to the second part of the proof of Lemma 2. □

Using the above lemma, we can show that the sufficient condition for FC-CBJ is similar to the sufficient condition for CBJ.

Theorem 6 *If a node is consistent and its parent is consistent with every set of variables, then FC-CBJ visits the node.*

Proof. Suppose that node $(a_1, \ldots, a_{i-1})$ is consistent with every set of variables, and its child $p = (a_1, \ldots, a_i)$ is consistent and not visited by FC-CBJ. Take the deepest j such that node $p' = (a_1, \ldots, a_j)$ is visited by FC-CBJ. Node p' is a proper ancestor of node p and is consistent with every set of variables. When FC-CBJ is at node p', none of the domains of the future variables is annihilated. The only reason for not instantiating the next variable x_{j+1} to a_{j+1} can be an A-type backtrack from some variable x_h to some variable x_g, where $g \leq j$ and $h \geq j + 2$. From Lemma 4 we know that the tuple composed of instantiations of the variables in the conflict set of x_h is inconsistent with some set of variables. Since the conflict set of x_h is a subset of $\{x_1, \ldots, x_g\}$ and $g < i$, this contradicts the initial assumption that $(a_1, \ldots, a_{i-1})$ is consistent with every set of variables. □

Now it is straightforward to prove the correctness of FC-CBJ.

Corollary 3 *FC-CBJ is correct.*

Proof. As in the proof of Corollary 2, the soundness is implied by the necessary condition, and the completeness by the sufficient condition. □

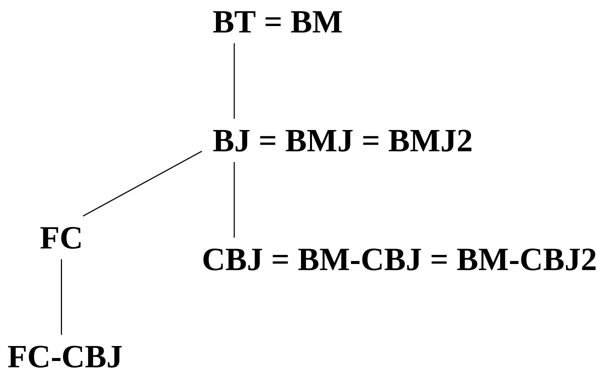

Figure 7: The hierarchy with respect to the number of visited nodes. Two algorithms are connected by an edge if the set of nodes visited by one of the algorithms is always a subset of the set of nodes visited by the other.

6 Hierarchies

We now present two hierarchies, which include the four basic backtracking algorithms discussed in Section 3, the Backmarking hybrids discussed in Section 4, and the FC-CBJ algorithm discussed in Section 5.

The hierarchy with respect to the number of visited nodes is presented in Figure 7. The relationships derived in Section 3 form the core of the hierarchy. Note that imposing a marking scheme on an algorithm does not change the set of nodes that are visited. Thus, for example, BM generates exactly the same backtrack tree as BT.

Figure 8 shows the hierarchy of algorithms with respect to the number of consistency checks. Since BT, BJ, and CBJ perform the same number of consistency checks at any given node, they are in the same order as in the nodes hierarchy. Imposing a marking scheme on a backtracking algorithm results in a reduction of the number of consistency checks performed.

Besides the relationships that are shown explicitly, it is important to note the ones that are implicit in the picture. In order to disprove a relationship between $\mathcal{A}$ and $\mathcal{B}$, one needs to find at least one constraint satisfaction problem on which $\mathcal{A}$ is better than $\mathcal{B}$, and one on which $\mathcal{B}$ is better than $\mathcal{A}$. For example, BM performs fewer consistency checks than FC on the regular 8-queens problem, but more on the confused 8-queens problem [13]. Examples of constraint networks were found that disprove all relationships that are not included in the hierarchies. Thus, however counterintuitive it may seem, FC-CBJ may visit more nodes than CBJ, and perform more consistency checks than BT.

The hierarchies are consistent with and explain some of the empirical results reported in the literature. For example, Prosser [16] compared how often one algorithm performed better than another with respect to consistency checks in a series of experiments to evaluate nine backtracking algorithms (Table 2, p. 293 in [16]). In this paper we have characterized eight of these nine backtracking algorithms (omitting FC-BJ). In half (14 out of 28) of the relevant pairwise comparisons, Prosser's experimental results showed that one algorithm always performed fewer consistency checks than the other. For 10 of these 14 cases, our theoretical results state that this must be the case. For the remaining 4 of these 14 cases, we have examples that show that this empirical result is not true in general. In particular, it is not necessarily the case that FC or FC-CBJ performs fewer consistency checks than BT or BJ.

The results presented in Sections 3–5 and summarized in the hierarchies shown in Figures 7 and 8 are stated and proven under the assumptions that the algorithms search for all solutions and that they instantiate the variables in a static ordering. We now relax both of

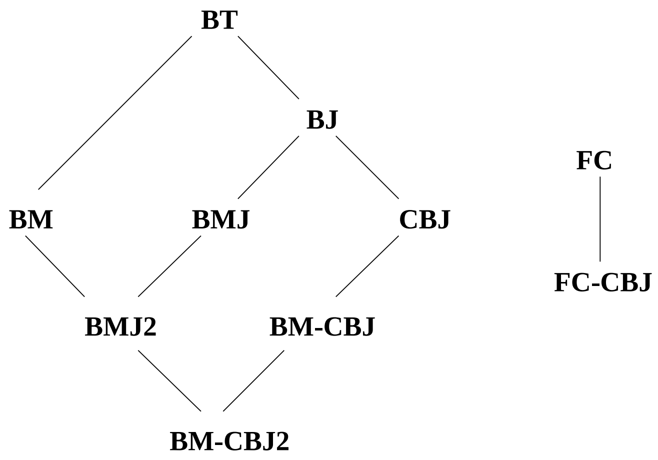

Figure 8: The hierarchy with respect to the number of consistency checks. Two algorithms are connected by an edge if one of the algorithms always performs no more consistency checks than the other.

these assumptions in turn and show that our results are still valid.

The assumption that the search is not interrupted until all possibilities are exhausted is not generally true if only a fixed number of solutions is sought. In order to deal with this issue, let us define two additional terms. Node p *precedes* node q in the search tree if $p = q$ or p is visited before q by the chronological backtracking algorithm (see the numbering on the nodes in Figure 6 for an example of such an ordering). Further, let the *termination node* be the last node visited by a backtracking algorithm. In the case of a backtracking algorithm that stops after finding the first solution, the termination node is either the first solution in the ordering, if a solution exists, or the last node in the ordering, if no solution exists. Now, we can reformulate the theorems to include an additional condition. For example, Theorem 1a would read:

a) If the parent of a node is consistent and the node precedes the termination node, then BT visits the node.

and Theorem 2a would read:

a) If BT visits a node, then its parent is consistent and the node precedes the termination node.

Given such a reformulation, the theorems and their corollaries can easily be proven without the assumption that all solutions are sought. It follows that our results also hold for the single solution versions of the algorithms.

We conclude this section with a discussion of the implications of our assumption of a static order of instantiation in which variables are added to the current partial solution according to the predefined order. With a static variable ordering, heuristics to order the variables may be used, but they must be applied before the constraint network is passed to a backtracking algorithm. A static order is in contrast to a dynamic order of instantiation in which the decision of which variable to instantiate next is based on the state of the search [2, 8]. Dynamic variable ordering (DVO) is known to be an effective technique. For example, Sabin and Freuder [17] specify a backtracking algorithm that maintains full arc consistency and performs DVO each time choosing the variable with the minimum remaining values (MRV) in its domain. They show experimentally that the algorithm performs very well on hard problems. Further, Bacchus and van Run [1] show that the forward checking algorithm equipped with the same DVO heuristic also performs very well on hard problems and on the benchmark Zebra and n-queens problems.

Our results are valid for the DVO versions of backtracking algorithms provided that the heuristic used for choosing the next variable is deterministic and independent of the backtracking algorithm. By independent we mean that the information exchanged between the heuristic and the backtracking algorithm is restricted as follows: only the constraint network and the partial solution are passed to the heuristic and only the next variable to instantiate is returned. In such a case, the choice of the next variable depends only on the state of the search and the backtracking algorithms will all make the same decision as to which variable to instantiate next given that they have reached the same node (partial solution). Thus, the ordering of the variables along any path from the root to a node will be identical and the nodes visited by the algorithms will continue to be a subset of the nodes visited by the BT algorithm that uses the same heuristic. The number of consistency checks performed by the algorithms will be uniformly increased by the number of checks performed by the heuristic, so that the consistency checks hierarchy will also remain unaffected. The results will not hold if, given the same constraint network and the same partial solution, the heuristic can return different answers on different invocations, such as would be the case, for example, if the algorithm broke ties randomly.

The assumption that the heuristic and the backtracking algorithm are independent is necessary for the results to hold for any DVO heuristic. However, given a particular heuristic we sometimes can relax the independence assumption in a principled way and still have our results hold. As an example of such an approach, let us consider the set of algorithms proposed by Bacchus and van Run [1]. They combine several backtracking algorithms, including the ones discussed in this paper, with a heuristic that at each node chooses the variable with the minimum remaining values (MRV). All backward checking algorithms, namely BT+MRV, BM+MRV, BJ+MRV, CBJ+MRV, and their hybrids, satisfy the condition of independence stated above; therefore, all partial order relationships between them remain valid. For FC+MRV and FC-CBJ+MRV, the condition is not satisfied because in both cases the algorithm and the heuristic share information through common data structures. However, since the direction of the flow of information is from the algorithm to the heuristic, the search tree remains unaffected, and consequently the node hierarchy is unchanged. Moreover, as the heuristic in both algorithms performs no additional consistency checks whatsoever, the relationship between FC+MRV and FC-CBJ+MRV is the same as between FC and FC-CBJ. Finally, the results stated in the hierarchies can be strengthened by including the result by Bacchus and van Run [1] that MRV makes standard backjumping redundant. Thus, in the node hierarchy BT+MRV = BM+MRV = BJ+MRV = BMJ+MRV = BMJ2+MRV and in the consistency check hierarchy BT+MRV = BJ+MRV and BM+MRV = BMJ+MRV = BMJ2+MRV.

7 Conclusions and Future Work

We presented a theoretical analysis of several backtracking algorithms. Such well-known concepts as backtrack, backjump, and domain annihilation were described in terms of inconsistency between instantiations and variables. This enabled us to formulate general theorems that fully or partially describe sets of nodes visited by the algorithms. The theorems were then used to prove the correctness of the algorithms and to construct hierarchies of algorithms with respect to the number of visited nodes and with respect to the number of consistency checks. The gaps in the resulting hierarchy prompted us to modify existing hybrid algorithms so that they are superior to the corresponding basic algorithms in every case. One of the modified algorithms is always better (in terms of consistency checks) than all six backward checking algorithms described by Prosser in [16].

There are several possible directions for future work. First, the sufficient and the necessary conditions are not identical for most of the algorithms investigated here. Since backtracking algorithms are deterministic, it may be possible to find single formulas that describe precisely their backtrack trees, as we did for BT and FC. Second, our approach could be applied to many other backtracking algorithms that have not been treated here, such as Dechter's graph-based backjumping algorithm [4] and Nadel's backtracking algorithm with full arc-consistency lookahead [13]. Finally, even though there is no absolute relationship between many pairs of algorithms, it may be possible to specify conditions under which such a relationship exists. For instance, one could try to specify formally the set of networks on which FC is always better than BT.

Acknowledgements

We would like to thank Dennis Manchak for his help in implementing the algorithms discussed in this paper and Fahiem Bacchus for helpful discussions on dynamic variable ordering and for comments on earlier versions of the paper. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

[1] F. Bacchus and P. van Run. Dynamic variable ordering in CSPs. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 258–275, Cassis, France, 1995. Available as: Springer Lecture Notes in Computer Science 976.

[2] J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Comm. ACM*, 18(11):651–656, 1975.

[3] M. Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12(1):36–39, 1981.

[4] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

[5] J. Gaschnig. A general backtracking algorithm that eliminates most redundant tests. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, page 457, Cambridge, Mass., 1977.

[6] J. Gaschnig. Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the 2nd Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 268–277, Toronto, Ont., 1978.

[7] M. L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[8] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[9] G. Kondrak. A theoretical evaluation of selected backtracking algorithms. Technical Report TR94–10, University of Alberta, June 1994.

[10] A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence, 2nd Edition*, pages 285–293. John Wiley & Sons, 1992.

[11] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inform. Sci.*, 19:229–250, 1979.

[12] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[13] B. A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[14] B. A. Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert*, 5(3):16–23, 1990.

[15] B. Nudel. Consistent-labeling problems and their algorithms: Expected-complexities and theory-based heuristics. *Artificial Intelligence*, 21:135–178, 1983.

[16] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[17] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 125–129, Amsterdam, 1994.

[18] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, 3:1–15, 1994.

[19] B. M. Smith and S. A. Grant. Sparse constraint graphs and exceptionally hard problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 646–651, Montreal, Que., 1995.

[20] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.