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
vanbeek@cs.ualberta.ca

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, for the first time, 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.

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 [1]. In recent years, many new backtracking algorithms have been proposed. The basic ones include Backmarking [3], Backjumping [4], Forward Checking [5], and Conflict-Directed Backjumping [10]. Several hybrid algorithms, which combine two or more basic algorithms, have also been developed [10].

There is no simple answer to the question 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 [9] ordered backtracking algorithms according to their

[*]Currently with MPR Teltech Ltd, Burnaby, BC, Canada

average-case performance. Prosser [10] 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 proven before.

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 [8]. Such algorithm, called BMJ, was presented by Prosser [10]. 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 [7] *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*[1] *(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.

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

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. A tuple is* consistent with a variable *if it is consistent with a one-element set containing this variable.*

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. In this work we assume the static *order of instantiation* in which variables are added to the current partial solution according to the predefined order: $x_1, \ldots, x_n$. 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 ϵ 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 lower levels, and the levels farther from the root are called higher levels. Similarly, the variables corresponding to these levels are called lower and higher. 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 lower 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.

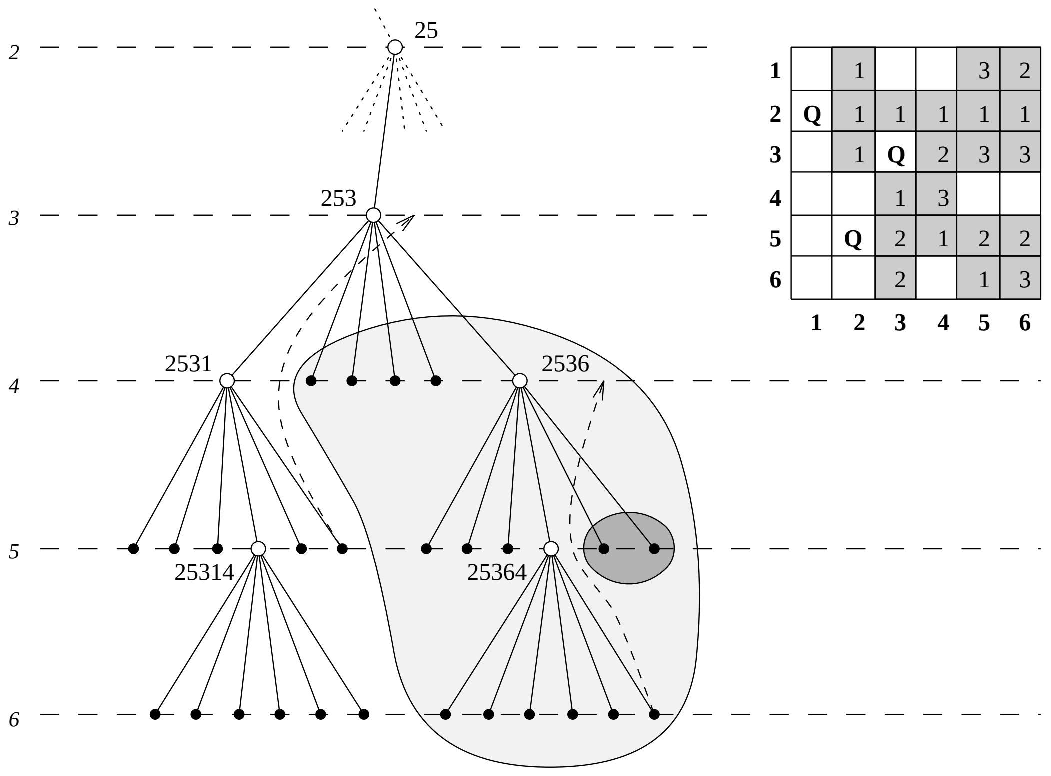
Chronological Backtracking (BT) [1] is the generic backtracking algorithm. The consistency checks between the instantiation of the current variable and the instantiations of the past variables are performed according to the original order of instantiations. If a consistency check fails, the next domain 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 when all consistency checks succeed after the last variable has been instantiated.

Backjumping (BJ) [4] is similar to BT, except that it behaves more efficiently when no consistent instantiation can be found for the current variable (at a dead-end). Instead of chronologically backtracking to the preceding variable, BJ backjumps to the highest past variable that was checked against the current variable.

Conflict-Directed Backjumping (CBJ) [10] 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 the instantiation a_i of the current variable and some past instantiation a_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 highest variable x_h in the conflict set of x_i. At the same time, the conflict set of x_i is absorbed by 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) [5] 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 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. Our representation of this problem identifies board columns with variables, and rows with domain values. Figure 1 shows a fragment of the backtrack tree generated by Chrono-



	1	2	3	4	5	6
1		1			3	2
2	**Q**	1	1	1	1	1
3		1	**Q**	2	3	3
4			1	3		
5		**Q**	2	1	2	2
6			2		1	3

Figure 1: A fragment of the BT backtrack tree for the 6-queens problem.

logical 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 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 highest 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), which is composed of 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 highest variable in the conflict set, which is x_5. No nodes are skipped at this point. The conflict set of x_6 is absorbed by 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 1 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, 2536 and 25314 are visited, but not 25364. The board in Figure 1 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 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. We deal with the more general problem of finding all solutions; at the end of the section we briefly comment on the validity of our results when only one solution is sought. The proofs that are not included here can be found in [6].

Lemma 1 *If BJ backtracks 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 highest variable in conflict with x_i. Let C_i denote the tuple composed of instantiations of all variables that are in conflict with x_i. Clearly, C_i is inconsistent with x_i. Since a_h is the instantiation of the highest variable in C_i, C_i is a subtuple of $(a_1, \ldots, a_h)$. Therefore, $(a_1, \ldots, a_h)$ is also inconsistent with x_i. □

Lemma 2 *If CBJ backtracks from variable x_i to variable x_h then C_i is inconsistent with S, where C_i is the tuple composed of instantiations of the variables in the conflict set of x_i, and S is a subset of $\{x_i, \ldots, x_n\}$ containing x_i.*

The following theorem specifies the sufficient conditions for a node to be visited by the four basic backtracking algorithms.

Theorem 1

a) BT visits a node if its parent is consistent.

b) BJ visits a node if its parent is consistent with all variables.

c) CBJ visits a node if its parent is consistent with all sets of variables.

d) FC visits a node if it is consistent and its parent is consistent with all variables.

Proof

b) Suppose that node $(a_1, \ldots, a_{i-1})$ is consistent with all variables, and its child $p = (a_1, \ldots, a_i)$ is not visited by BJ. Take the highest 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 all variables. 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 all variables.

c) Similar to the proof of b), except that we use Lemma 2.

Proofs of the remaining cases are straightforward. □

The next theorem specifies the necessary conditions for a node to be visited by the four backtracking algorithms.

Theorem 2

a) BT visits a node only if its parent is consistent.

b) BJ visits a node only if its parent is consistent.

c) CBJ visits a node only if its parent is consistent.

d) FC visits a node only if it is consistent and its parent is consistent with all variables.

Figure 2 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 2.

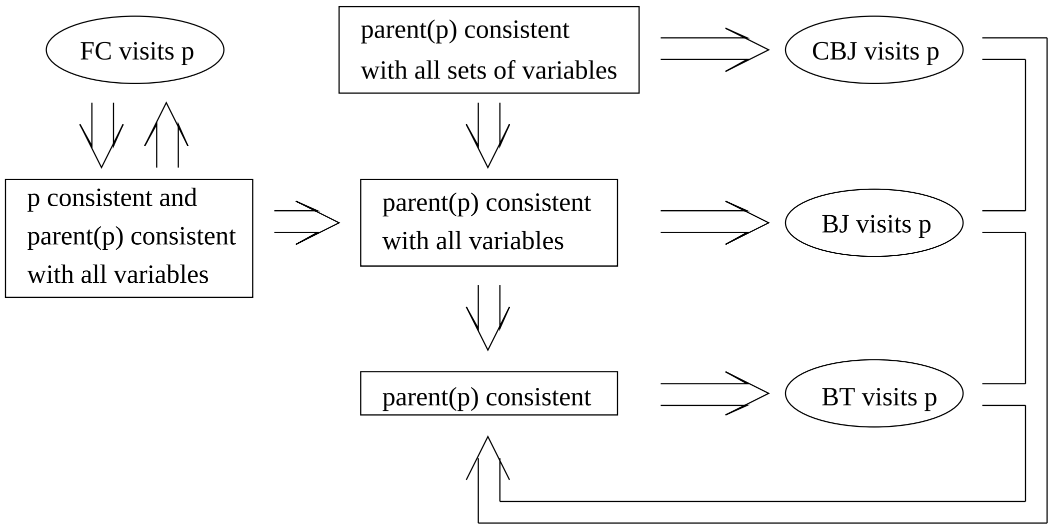

FC visits p
parent(p) consistent with all sets of variables
CBJ visits p
p consistent and parent(p) consistent with all variables
parent(p) consistent with all variables
BJ visits p
parent(p) consistent
BT visits p

Figure 2: Conditions graph.

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 the two lemmas from Section 3.

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

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 1 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 its parent $(a_1, \ldots, a_{n-1})$ 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 all variables. 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. □

To the best of our knowledge, this is the first time that BJ and CBJ have been proven to be correct[2]. Naturally, our approach can be extended to other backtracking algorithms.

All the above results were originally proven with the assumption that the search is not interrupted until all possibilities are exhausted. This is not generally true if only a fixed number of solutions is sought. However, if we restrict our attention to only those of the search tree nodes that precede (in the preorder

[2]Backjumping was first presented without formal proof and the suggested future work was to prove that it is a valid algorithm [11]. Recently, a reviewer of [10] commented: "I am not convinced that BJ or CBJ are sound! [Gaschnig's original algorithm] was never proved to be correct" [11].

traversal) the last node visited by a backtracking algorithm, the theorems are still valid. Therefore, our results hold also for the "one solution" versions of the backtracking algorithms, with only slightly modified proofs.

4 Hybrid Algorithms with Backmarking

In this section we briefly discuss Backmarking [3] and its two hybrids. We propose a modification to the hybrids, and then include these algorithms in our hierarchies.

Backmarking (BM) imposes a marking scheme on the Chronological Backtracking algorithm in order to eliminate some redundant consistency checks. The scheme is based on the following two observations [8]: (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 marking scheme is implemented using two arrays: *mbl* (minimum backup level) of size n, and *mcl* (maximum checking level) of size $n \times m$. The entry *mbl*[i] contains the number of the lowest 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 highest variable that was checked against the j-th value in the domain of the variable x_i.

Nadel [8] suggested combining BM and BJ into a new hybrid algorithm. Prosser presented such algorithm, called Backmarking and Backjumping (BMJ), in [10]. 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.

A careful analysis of the algorithm leads us to the conclusion that BMJ is sometimes worse than BM because the *mbl* array, which was originally designed for a chronologically backtracking algorithm, is no longer adequate for a backjumping algorithm. Since BM always tests *all* values of the current variable for consistency, a single entry for all values is sufficient. In BMJ, however, it often happens that only *some* values of the current instantiation are tested, and the other values are skipped by a backjump. A separate entry for each value is therefore necessary to preserve all collected consistency information.

We propose a modified BackMarkJump (BMJ2),

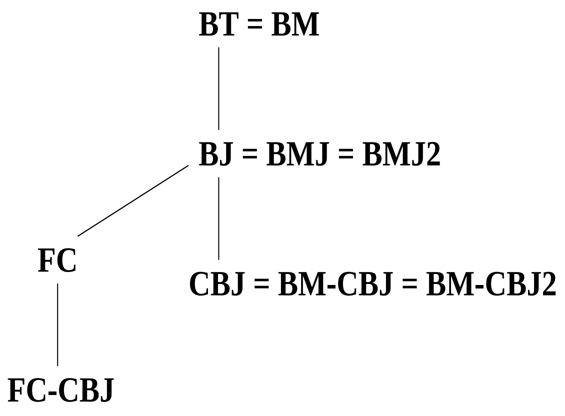


Figure 3: The hierarchy with respect to the number of visited nodes.

which solves the problem by making *mbl* a two-dimensional rather than a one-dimensional array. The new *mbl* array is of size $n \times m$, where n is the number of variables, and m is the size of the largest domain. This is a reasonable space requirement because BMJ already uses one $n \times m$ array; each *mcl* entry has now a corresponding *mbl* entry. The *mbl*[i][j] entry stores the number of the lowest variable whose instantiation has changed since the variable x_i was last instantiated with the j–th value. The entry is set to i every time the current instantiation (x_i, t_j) is being tested for consistency with past instantiations. When the algorithm backtracks, the entries are updated in a similar way as in BMJ. Thanks to the more efficient backmarking scheme BMJ2 is always better than BMJ. Moreover, since BMJ2 does not lose information about consistency checks in the way BMJ does, it is always better than BM.

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 Hierarchies

We now present two hierarchies, which include the four basic backtracking algorithms described in Section 2, and the Backmarking hybrids discussed in Section 4.

The hierarchy with respect to the number of visited nodes is presented in Figure 3. Two algorithms are connected by a link if the set of nodes visited by one of them is always a subset of the set of nodes visited by the other. Naturally, the relation is transitive. The relationships derived in Section 3 form the core of the hierarchy. Note that imposing a backmarking 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 4 shows the hierarchy of algorithms with

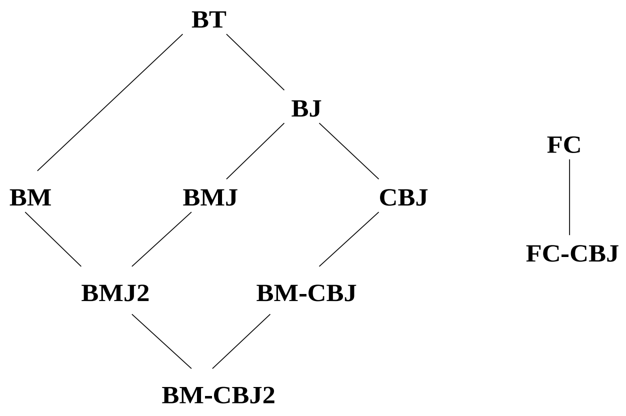


Figure 4: The hierarchy with respect to the number of consistency checks.

respect to the number of consistency checks. Two algorithms are connected by a link if one of them always performs no more consistency checks than the other. 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. The figure contains also one Forward Checking hybrid: Forward Checking and Conflict-Directed Backjumping (FC-CBJ) [10], which has not been discussed here. For a treatment of FC-CBJ see [6].

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 [8]. 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.

6 Conclusions

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 [10]. In the future the hierarchies could be extended by applying our approach to other backtracking algorithms, such as Dechter's graph-based backjumping algorithm [2] and Nadel's backtracking algorithm with full arc-consistency lookahead [8].

References

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

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

[3] J. Gaschnig. A general backtracking algorithm that eliminates most redundant tests. In *Proc. of the Int'l Joint Conf. on Artificial Intelligence*, page 457, 1977.

[4] J. Gaschnig. Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisficing assignment problems. In *Proc. of the 2nd Biennial Conf. of the Canadian Soc. for Comput. Studies of Intell.*, pages 268–277, 1978.

[5] R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–314, 1980.

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

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

[8] B. Nadel. Constraint satisfaction algorithms. *Comput. Intell.*, 5:188–224, 1989.

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

[10] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Comput. Intell.*, 9(3):268–299, 1993.

[11] P. Prosser. Personal communication, 1994.