UNIVERSITY OF ALBERTA

A Theoretical Evaluation of Selected Backtracking Algorithms

BY

Grzegorz Kondrak

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Fall 1994

Abstract

In recent years, numerous new backtracking algorithms have been proposed. The algorithms are usually evaluated by empirical testing. This method, however, has its limitations. Our thesis adopts a different, purely theoretical approach, which is based on characterizations of the sets of search tree nodes visited by the backtracking algorithms. A new notion of inconsistency between instantiations and variables is introduced, a useful tool for describing 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 visited nodes, and the number of performed consistency checks. 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 less consistency checks than the original algorithms.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Constraint-based reasoning is a simple, yet powerful paradigm in which many interesting problems can be formulated. A constraint network is defined by a set of variables, a domain of values for each variable, and a set of constraints between the variables. The area of constraint-based reasoning has received a lot of attention recently, and numerous methods for dealing with constraint networks have been developed. The applications of constraint networks include graph coloring, scene labelling, natural language parsing, and temporal reasoning.

Constraint networks can be solved using backtracking search. The generic backtracking algorithm was first described more than a century ago, and since then has been re-discovered many times [2]. In recent years, numerous new backtracking algorithms have been proposed. The basic ones include Backjumping [7], Conflict-Directed Backjumping [18], Graph-Based Backjumping [4], Backmarking [6], and Forward Checking [10]. Several hybrid algorithms, which combine two or more basic algorithms, have also been developed [18, 19].

A question arises as to which of the known backtracking algorithms is the best one. There is no straightforward answer. 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 the purpose of 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 which verify that the current instantiations of two variables satisfy the constraints) cannot be determined in abstraction from a concrete problem. The number of consistency checks seems to be a more legitimate measure of the efficiency of a backtracking algorithm, although it neglects the "overhead" costs incurred by maintaining sophisticated data structures. Another standard measure is the number of nodes in the backtrack tree generated by an algorithm.

Prosser, in his landmark technical report[1] [17], presents nine backtracking algo-

[1]It has also appeared recently as a journal article [20].

rithms in a uniform notation, thus facilitating their comparison. Prosser performed a series of experiments to evaluate the algorithms against each other. Table 1.1 shows how often one algorithm performed less consistency checks than another over 450 instances of the zebra problem, a well-known benchmarking problem. The entries containing zeros are especially interesting because they may indicate that one algorithm is *always* better than another. However, such a hypothesis can never be verified solely by experimentation; the relationship has to be proven theoretically. In fact, in the following chapters, it will be shown that some of the zero entries indicate a general rule, whereas other do not.

	BT	BJ	CBJ	BM	BMJ	BM-CBJ	FC	FC-BJ	FC-CBJ
BT	-	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
BJ	450	-	**0**	132	**0**	**0**	**0**	**0**	**0**
CBJ	450	450	-	370	280	**0**	130	35	5
BM	450	318	80	-	31	8	13	5	2
BMJ	450	450	170	419	-	17	29	7	3
BM-CBJ	450	450	450	442	433	-	286	117	35
FC	450	450	320	437	421	163	-	**0**	**0**
FC-BJ	450	450	415	445	443	333	438	-	**0**
FC-CBJ	450	450	445	448	447	415	440	388	-

Table 1.1: How often one algorithm bettered another [17].

In this work we adopt a purely theoretical approach. We analyze several backtracking algorithms with the purpose of discovering general rules that determine their behaviour. A new notion of inconsistency between instantiations and variables is introduced, a useful tool for describing such well-known concepts as backtrack, backjump, and domain annihilation. For every algorithm we attempt to formulate the necessary and sufficient conditions for a search tree node to be visited by the algorithm. Sometimes both conditions are the same, which gives us a complete characterization of the set of visited nodes. More often we have to make do with a partial characterization, which leaves out a "grey zone" of nodes that may or may not be visited by the algorithm.

The characterizing conditions enable us to: (a) prove the correctness of the algorithms, and (b) construct partial orders (or *hierarchies*) of the algorithms according to standard performance measures. 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.

The proofs are independent of the implementation method. We do not prove the correctness of every backtracking algorithm discussed in this work, but rather present a methodology which can be applied to any backtracking algorithm. All proofs presented here are original. We hope that, apart from demonstrating correctness, our new approach will provide a deeper understanding of how the algorithms work. Such insight may result in less time spent on the implementation and debugging of the algorithms.

Hierarchies may be useful to anyone who is faced with a choice of a backtracking algorithm. With so many backtracking algorithms around, it is difficult to implement and test all of them. The hierarchies make the selection of the right algorithm easier once it has been established what one's priorities are. For example, someone may be interested solely in reducing the number of consistency checks, while for someone else the complexity of the code may be the main factor. We present two hierarchies; one orders the algorithms according to the number of visited nodes, and the other according to the number of performed consistency checks.

The need for hierarchies has been recognized before. Nudel [16] gives a ranking of some backtracking algorithms based on the average-case performance reported by Haralick [10]. Prosser [17] orders nine backtracking algorithms according to their average-case performance on 450 instances of the zebra problem. However, such an approach is open to the criticism that the test problems are not representative of the problems that arise in practice[2]. Even a theoretical average-case analysis is possible only if one makes simplifying assumptions about the distribution of problems. In contrast, our hierarchies are valid for all instances of all constraint satisfaction problems.

In the conclusion of his paper which presented the new hybrid backtracking algorithms, Prosser posed the following question [17]:

> 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 less consistency checks than the corresponding basic algorithms. It is important that hybrid algorithms have this property in order to offset the disadvantages of a more complex code and higher overhead costs.

The thesis is organized as follows. Chapter 2 contains the necessary definitions and the descriptions of the backtracking algorithms. Chapter 3 shows our methodology applied to four basic backtracking algorithms. Chapter 4 extends the approach to other backtracking algorithms. Chapter 5 presents some experimental results. Chapter 6 provides suggestions for future work and a summary.

[2]Prosser acknowledges this in [17]: "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."

Chapter 2

Background

This chapter contains the necessary definitions and the descriptions of the backtracking algorithms. In the first section, a new notion of inconsistency between instantiations and variables is introduced. The basic concepts of the constraint satisfaction paradigm are also included. In the second section, backtracking algorithms are identified by presenting the C-language code adapted from a CSP function library [12].

2.1 Definitions

We begin with some basic concepts of the constraint satisfaction paradigm.

Definition 1 *A* binary constraint network*[13] 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, t) assigns the value t 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 $((x_1, X_1), \ldots, (x_i, X_i)))$ to $(X_1, \ldots, X_i)$.*

In this work we introduce a new notion of consistency between a tuple of instantiations and a set of variables. This notion is fundamental to virtually all proofs presented in this work.

Definition 2 *A tuple $((x_{i_1}, X_{i_1}), \ldots, (x_{i_u}, X_{i_u}))$ is* consistent with a set of variables *$\{x_{j_1}, \ldots, x_{j_v}\}$ if there exist instantiations $X_{j_1}, \ldots, X_{j_v}$ of the variables $x_{j_1}, \ldots, x_{j_v}$ respectively, such that the tuple $((x_{i_1}, X_{i_1}), \ldots, (x_{i_u}, X_{i_u}), (x_{j_1}, X_{j_1}), \ldots, (x_{j_v}, X_{j_v}))$ is*

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

consistent. A tuple is consistent with a variable *if it is consistent with a one-element set containing this variable. Instead of writing* "is consistent with", *we sometimes use the symbol* Δ.

The notion of consistency between a tuple and a set of variables can also be expressed by the following formula:

$$((x_{i_1}, X_{i_1}), \ldots, (x_{i_u}, X_{i_u})) \;\Delta\; (S \cup \{x_j\}) \equiv \\ \exists t \in D_j : ((x_{i_1}, X_{i_1}), \ldots, (x_{i_u}, X_{i_u}), (x_j, t)) \;\Delta\; S, \tag{2.1}$$

where S is a set of variables.

By applying the above formula n times[2] we obtain:

$$\epsilon \;\Delta\; \{x_1, \ldots, x_n\} \equiv \\ \exists t_1 \in D_1 : \ldots \exists t_n \in D_n : ((x_1, t_1), \ldots, (x_n, t_n)) \;\Delta\; \emptyset, \tag{2.2}$$

where ϵ is the empty tuple. Informally, the equation states that a network of n variables is consistent if and only if there exists a solution to the network.

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 [15]). 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 $R_{ij} = \{(X_i, X_j) : (X_i \neq X_j) \wedge (|i - j| \neq |X_i - X_j|)\}$. Figure 2.1 shows two instances of the 4-queens problem. The shaded squares denote the positions which are excluded from consideration by the already placed queens. 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 ϵ is also consistent with all sets of variables.

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 of instantiation*: $x_1, \ldots, x_n$. It is convenient to divide all variables into three sets:

[2]Throughout the thesis we use n to denote the number of variables in the network.

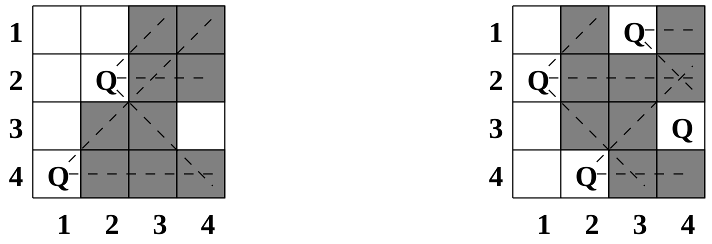


Figure 2.1: A partial and a full solution to the 4-queens problem.

- *past variables* – those which have already been instantiated,
- *current variable* – that which is being instantiated,
- *future variables* – those which are to be instantiated.

A *dead-end* is the situation 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* or *backstepping*. Otherwise, it is *backjumping*. A backtracking algorithm terminates when all possible assignments have been tested or a certain number of solutions have been found.

Since we make extensive use of tree terminology, a few definitions are in order (see [1]). A *tree* is a directed graph with no cycles satisfying the following properties:

1. There is exactly one node, called the *root*, which no edges enter.
2. Every node except the root has exactly one entering edge.
3. There is a path from the root to each node.

If there is an edge from node v to node w then v is called the *parent* of w, and w is a *child* of v. If there is a path from v to w then v is an *ancestor* of w and w is a *descendant* of v. Furthermore, if $v \neq w$ then v is a *proper ancestor* of w, and w is a *proper descendant* of v. A node with no proper descendants is called a *leaf*. A node v and all its descendants are called a *subtree*. The node v is called the *root* of that subtree. The *level of a node* v in a tree is the length of the path from the root to v.

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 which represent consistent tuples are called *consistent* *nodes*. The nodes which 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 instantiations of the current variable and 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 (closer to the root) levels of the search tree. Whenever some variables are uninstantiated and x_h is set as the new current variable, we say that the algorithm backtracks to level h.

Example 2. The confused n-queens problem, described in [14], is how to place n queens on a $n \times n$ chess board, one queen per column, so that all queens *do* attack each other. Similarly to the regular n-queens problem, 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 $R_{ij} = \{(X_i, X_j) : (X_i = X_j) \vee (|i-j| = |X_i - X_j|)\}$. Figure 2.2 shows the search tree for the confused 3-queens problem. Horizontal dashed lines represent levels of the search tree, which correspond to variables. White dots denote consistent nodes. Black dots denote inconsistent nodes. The circled consistent nodes at the last level of the tree are the solution nodes. Nodes are labelled according to the tuples they represent, but parentheses and commas have been omitted for clarity; for instance, node '23' represents tuple (2,3). All nodes except the six nodes marked with 'x' belong to the backtrack tree of the generic backtracking algorithm (BT).

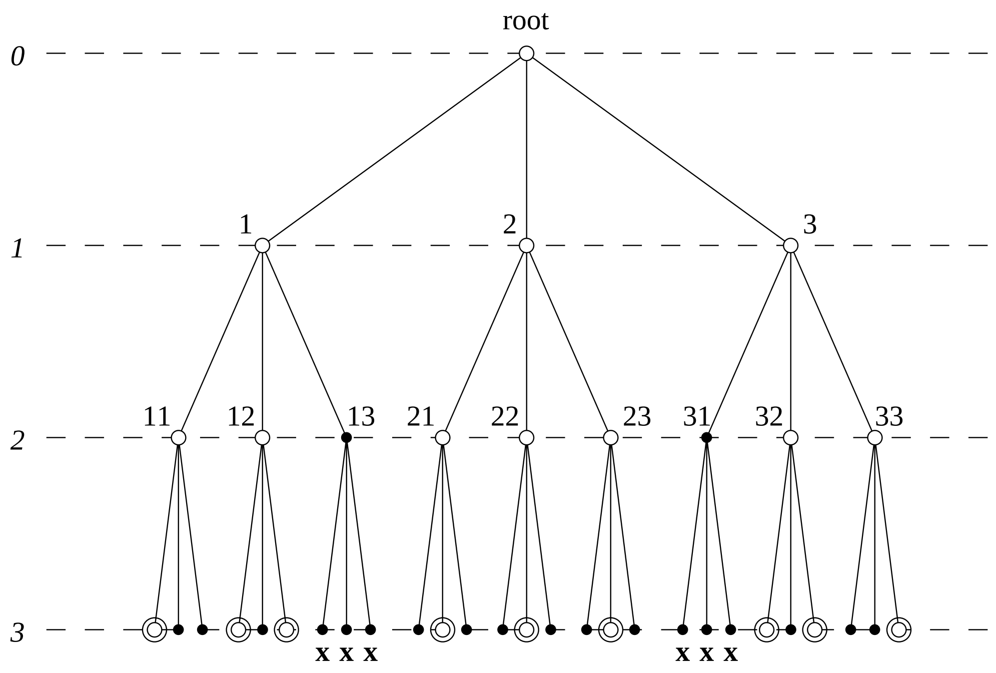


Figure 2.2: An example of a search tree.

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.

2.2 Backtracking Algorithms

In this section we present six basic backtracking algorithms and four of Prosser's hybrid backtracking algorithms. We identify the algorithms by including the C language source code from a CSP library [12]. We chose to include the C code rather than pseudocode because the C language syntax is widely known and unambiguous. The code can be actually compiled and run, given suitable header files. The names of the algorithms are the same as in [17]. These versions of the algorithms are designed to find the first solution only.

It must be stressed that the algorithms may be implemented in many different ways. It is important, however, that all implementations of the same algorithm generate the same backtrack tree and perform the same consistency checks.

2.2.1 Functions and Data Structures

The following variables, constants and routines are used:

- N is a constant that denotes the number of variables.
- K is a constant that denotes the domain size (for simplicity it is assumed that all domains have the same size).
- *current* is a variable that contains the number of the current variable.
- v is a one-dimensional array of size N that contains the current instantiations of the variables.
- The main function of every algorithm returns the number of the variable which is selected as the backtracking point (in some cases the return value is not used).
- Function *consistent(current)* returns 1 if the current instantiation is consistent with past instantiations (or, in the case of forward checking algorithms, future variables), and 0 otherwise.
- Function *check(i,j)* returns 1 if the consistency check between $v[i]$ and $v[j]$ succeeds, and 0 otherwise.
- Procedure *solution()* processes the solution stored in the v array (if only one solution is sought, it also terminates the algorithm).
- Procedure *merge(*S_1*,*S_2*)* merges two sets: $S_1 := S_1 \cup S_2$.
- Procedure *empty(S)* empties a set: $S := \emptyset$.

- Procedure *add(x,S)* adds an element to a set: $S := S \cup \{x\}$.
- Procedure *delete(x,S)* deletes an element from a set $S := S - \{x\}$.
- Function *max(S)* returns the maximal element of set S.

The search is started by invoking the main function with the first variable as the parameter.

2.2.2 Chronological Backtracking (BT)

Chronological Backtracking (BT) [9, 2] is the generic backtracking algorithm. This is the starting point for all modifications that result in more sophisticated backtracking algorithms. The main advantage of BT is its simplicity. It always backtracks chronologically to the most recently instantiated variable. While BT is more efficient than the naive "generate and test" approach, there is still much room for improvement.

```
int consistent(current)
int current;
{
    int i;

    for (i = 1; i < current; i++)
        if (check(current,i) == 0)
            return(0);
    return(1);
}

int BT(current)
int current;
{
    int i;

    if (current > N) {
        solution();
        return(N); }
    for (i = 0; i < K; i++) {
        v[current] = i;
        if (consistent(current))
            BT(current + 1); }
    return(current-1);
}
```

In terms of the backtrack tree the algorithm can be described as follows. When an i-level node is visited, consistency checks are performed between the instantiation of the current variable and all earlier instantiations along the corresponding branch of the tree, starting from level 1. If all checks succeed, the branch is extended by instantiating the next variable x_{i+1} to each of the values in its domain. Otherwise the branch is abandoned, and the next domain value is tried. If there are no more values to be tried for the current variable, BT backtracks to level $i-1$. A solution is recorded every time when all consistency checks succeed at an n-level node.

BT often generates subtrees that are identical to previously explored subtrees by instantiating variables that play no role in the current inconsistency. Such behaviour is called *thrashing* [11]. Other backtracking algorithms attempt to minimize thrashing.

2.2.3 Backjumping (BJ)

Backjumping (BJ) [7] is similar to BT, except that it behaves more efficiently when no consistent instantiation can be found for the current variable (a dead-end). Instead of backstepping to the preceding variable, BJ backjumps to the highest variable that conflicted with the current variable.

```
int consistent(current)
int current;
{
    int i;

    for (i = 1; i < current; i++)
        if (check(current,i) == 0) {
            max_check[current] = max(max_check[current],i);
            return(0); }
    max_check[current] = current - 1;
    return(1);
}

int BJ(current)
int current;
{
    int i, jump;

    if (current > N) {
        solution();
        return(N); }
    max_check[current] = 0;
    for (i = 0; i < K; i++) {
        v[current] = i;
        if (consistent(current)) {
            jump = BJ(current + 1);
            if (jump != current)
                return(jump); } }
    return(max_check[current]);
}
```

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 $(x_1, x_2, \ldots)$. The checking stops as soon as one consistency check fails. The entry *max_check*$[i]$ stores the number of the highest variable that was checked against the current instantiation of x_i. This value is used for determining the backtracking point at the end of the main function.

The main problem with BJ is that it backjumps only from dead-ends. All other backtracks are chronological, so there is still a lot of thrashing. On the plus side, the overhead costs are small in BJ.

2.2.4 Conflict-Directed Backjumping (CBJ)

Conflict-Directed Backjumping (CBJ) [18] has even 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 X_i of the current variable and some past instantiation 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 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.

```
int consistent(current)
int current;
{
    int i;

    for (i = 1; i < current; i++)
        if (check(current,i) == 0) {
            add(i,conf_set[current]);
            return(0); }
    return(1);
}

int CBJ(current)
int current;
{
    int h, i, jump;

    if (current > N) {
        solution();
        return(N); }
    empty(conf_set[current]);
    for (i = 0; i < K; i++) {
        v[current] = i;
        if (consistent(current)) {
            jump = CBJ(current + 1);
            if (jump != current)
                return(jump); } }
    h = max(conf_set[current]);
    merge(conf_set[h], conf_set[current]);
    return(h);
}
```

The code of CBJ is similar to BJ. Instead of the simple array *max_check* we have an array of sets *conf_set*. At a dead-end, *max_check*[*i*] corresponds to the maximal element of *conf_set[i]*.

CBJ has an ability to perform "multiple backjumps," that is, after the initial backjump from a dead-end it can continue backjumping across conflicts, which may potentially result in significant savings. This comes at a price, however, because the cost of maintaining additional data structures is higher than in BJ.

2.2.5 Graph-Based Backjumping (GBJ)

Graph-Based Backjumping (GBJ) [4] similarly attempts to backtrack more than one level if possible. It utilizes knowledge about the constraint graph to backtrack to the highest variable connected [3] to the current one.

```
int consistent(current)
int current;
{
    int i;

    for (i = 1; i < current; i++) {
        if (check(current,i) == 0)
            return(0); }
    return(1);
}

int GBJ(current)
int current;
{
    int h, i, jump;

    if (current > N) {
        solution();
        return(N); }
    for (i = 0; i < K; i++) {
        v[current] = i;
        if (consistent(current)) {
            jump = GBJ(current + 1);
            if (jump != current)
                return(jump); } }
    merge(P,parents(current));
    h = max(P);
    delete(h, P);
    return(h);
}
```

Function *consistent*() is the same as in BT. There are no additional operations during consistency checking. Function *parents*(i) returns the PARENTS set of x_i — the set of variables connected to x_i that precede it in the instantiation order. For example, in the constraint network shown in Figure 4.10: PARENTS[x_1] = $\emptyset$, PARENTS[x_2] = $\emptyset$, PARENTS[x_3] = $\{x_1, x_2\}$, PARENTS[x_4] = $\{x_2\}$. P is a global set variable (initially empty) which contains variables that may have caused the inconsistency.

GBJ is significantly better than BT only if the constraint graph is sparse. If the constraint graph is fully connected, GBJ generates the same backtrack tree as BT. The overhead costs are smaller than in CBJ because the PARENTS sets need only be computed once, before the search begins.

[3] Two variables are connected if there is a nontrivial constraint between them.

2.2.6 Backmarking (BM)

Backmarking (BM) [6] 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 [14]: (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 (type-A savings). (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 (type-B savings).

```
int consistent(current)
int current;
{
    int i, oldmbl;

    oldmbl = mbl[current];
    if (mcl[current][v[current]] < oldmbl)
        return(0);
    for (i = oldmbl; i < current; i++) {
        mcl[current][v[current]] = i;
        if (check(current,i) == 0)
            return(0); }
    return(1);
}

int BM(current)
int current;
{
    int h, i;

    if (current > N) {
        solution();
        return(N); }
    for (i = 0; i < K; i++) {
        v[current] = i;
        if (consistent(current))
            BM(current + 1); }
    h = current - 1;
    mbl[current] = h;
    for (i = h+1; i <= N; i++)
        mbl[i] = min(mbl[i],h);
    return(h);
}
```

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. All entries in both variables are initially set to 1. The behaviour of BM will be analyzed in detail in Section 4.1.

BM visits exactly the same nodes as BT, with all the thrashing involved. However, at some nodes it may perform no consistency checks at all.

2.2.7 Forward Checking (FC)

So far, all described algorithms perform the consistency checks backward, that is, between the current variable and the past variables. For this reason we call them the *backward checking* algorithms. In contrast, Forward Checking (FC) [10] performs consistency checks forward, that is, between the current variable and the future variables. When an i-level node is visited, 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 branch is extended by instantiating the next variable x_{i+1} to each of the values in its filtered domain. Otherwise, the branch is abandoned, the effects of forward checking are undone, and the next value is tried. If there is no more values to tried for the current variable, FC backs up to the level $i-1$. A solution is recorded every time an n-level node is reached.

```
void restore(i)
int i;
{
    int j, a;

    for (j = i+1; j <= N; j++)
        if (checking[i][j]) {
            checking[i][j] = 0;
            for (a = 0; a < K; a++)
                if (domains[j][a] == i)
                    domains[j][a] = 0; }
}

int consistent(current)
int current;
{
    int j, a;
    int old = 0, del = 0;

    for (j = current + 1; j <= N; j++) {
        for (a = 0; a < K; a++)
            if (domains[j][a] == 0) {
                old++;
                v[j] = a;
                if (check(current,j) == 0) {
                    domains[j][a] = current;
                    del++; } }
        if (del)
            checking[current][j] = 1;
        if (old - del == 0)
```

```
            return(j); }
    return(0);
}

int FC(current)
int current;
{
    int i, fail;

    if (current > N) {
        solution();
        return(N); }
    for (i = 0; i < K; i++) {
        if (domains[current][i])
            continue;
        v[current] = i;
        fail = consistent(current);
        if (fail == 0)
            FC(current + 1);
        restore(current); }
    return(current-1);
}
```

FC uses two global arrays. The integer array *domains* is of size $N \times K$. If $domains[i][j] = t$ and $t > 0$, it means that the j–th value has been removed from the domain of variable x_i because of the current instantiation of the variable x_t. If $t = 0$, the value is still in the domain. The boolean array *checking* is of size $N \times N$. The entry $checking[i][j]$ is set if the current instantiation of variable x_i causes removal of some value from the domain of future variable x_j. Otherwise, it is cleared. All entries in both arrays are initially set to 0.

Forward consistency checking is handled by two routines. Function $consistent(i)$ returns the number of the variable which has been annihilated during forward checking. If no variable has been annihilated, the function returns 0. Procedure $restore(i)$ undoes the changes caused by the instantiation of x_i.

FC is very efficient because of its ability to discover inconsistencies early. The size of the backtrack tree is thus greatly reduced. However, since it consults all variables in the network after every new instantiation, FC sometimes performs consistency checks that are avoided by the backward checking algorithms.

2.2.8 Backmarking Hybrids

Backmarking and Backjumping (BMJ), and Backmarking and Conflict-Directed Backjumping (BM-CBJ) [17] incorporate backjumping within the Backmarking algorithm. Both algorithms are similar to BM. The difference lies in using additional data structures for backjumping: *max_check* in BMJ, and *conf_set* in BM-CBJ. The code of both algorithms is presented side by side in order to emphasize their similarity. Only the lines marked by '#' are different. The lines marked by '=' are identical.

```
int consistent(z)                    = int consistent(z)
int z;                               = int z;
{                                    = {
  int i, oldmbl;                     =   int i, olmbl;
                                     =
  oldmbl = mbl[z];                   =   oldmbl = mbl[z];
  if (mcl[z][v[z]] < oldmbl) {       =   if (mcl[z][v[z]] < oldmbl) {
                                     #     add(mcl[z][v[z]],conf_set[z]);
    return(0); }                     =     return(0); }
  for (i = oldmbl; i < z; i++) {     =   for (i = oldmbl; i < z; i++) {
    mcl[z][v[z]] = i;                =     mcl[z][v[z]] = i;
    if (check(z,i) == 0) {           =     if (check(z,i) == 0) {
      max_check[z] =                 #       add(i,conf_set[z]);
          max(max_check[z],i);       #
      return(0); } }                 =       return(0); } }
  max_check[z] = z - 1;              #
  return(1);                         =   return(1);
}                                    = }
                                     =
int BMJ(z)                           # int BM_CBJ(z)
int z;                               = int z;
{                                    = {
  int h, i, jump;                    =   int h, i, jump;
                                     =
  if (z > N) {                       =   if (z > N) {
    solution();                      =     solution();
    return(N); }                     =     return(N); }
  max_check[z] = 0;                  #   empty(conf_set[z]);
  for (i = 0; i < K; i++) {          =   for (i = 0; i < K; i++) {
    v[z] = i;                        =     v[z] = i;
    if (consistent(z)) {             =     if (consistent(z)) {
      jump = BMJ(z + 1);             =       jump = BM_CBJ(z + 1);
      if (jump != z)                 =       if (jump != z)
        return(jump); } }            =         return(jump); } }
  h = max_check[z];                  #   h = max(conf_set[z]);
                                     #   merge(conf_set[h],conf_set[z]);
  mbl[z] = h;                        =   mbl[z] = h;
  for (i = h+1; i <= N; i++)         =   for (i = h+1; i <= N; i++)
    mbl[i] = min(mbl[i],h);          =     mbl[i] = min(mbl[i],h);
  return(h);                         =   return(h);
}                                    = }
```

It is straightforward to create other hybrids: BM-GBJ, which combines BM and GBJ, and FC-GBJ, which combines FC and GBJ. We will not, however, discuss GBJ hybrids in this work, because they are neither conceptually simple nor more efficient than the already known backtracking algorithms.

2.2.9 Forward Checking Hybrids

Forward Checking and Backjumping (FC-BJ) and Forward Checking and Conflict-Directed Backjumping (FC-CBJ) [17] incorporate backjumping within the Forward Checking algorithm. In contrast with FC, which always backtracks chronologically, the FC hybrids record the information about the variables that caused the current inconsistency. Later, this information is used to determine the backtracking point.

```
int FC_BJ(z)                         #  int FC_CBJ(z)
int z;                               =  int z;
{                                    =  {
  int h, i, j, jump, fail;           =    int h, i, j, jump, fail;
                                     =
  if (z > N) {                       =    if (z > N) {
    solution();                      =      solution();
    return(N); }                     =      return(N); }
  max_check[z] = 0;                  #    empty(conf_set[i]);
  for (i = 0; i < K; i++) {          =    for (i = 0; i < K; i++) {
    if (domains[z][i])               =      if (domains[z][i])
      continue;                      =        continue;
    v[z] = i;                        =      v[z] = i;
    fail = consistent(z);            =      fail = consistent(z);
    if (fail == 0) {                 =      if (fail == 0) {
      max_check[z] = z-1;            #
      jump = FC_BJ(z + 1);           #        jump = FC_CBJ(z + 1);
      if (jump != z)                 =        if (jump != z)
        return(jump); }              =          return(jump); }
    restore(z);                      =      restore(z);
    if (fail)                        =      if (fail)
      for (j = 1; j < z; j++)        =        for (j = 1; j < z; j++)
        if (checking[j][fail])       =          if (checking[j][fail])
          max_check[z] =             #            add(j,conf_set[z]); }
          max(max_check[z],j);}      #
  h = max_check[z];                  #
  for (j = 1; j < z; j++)            =    for (j = 1; j < z; j++)
    if (checking[j][z])              =      if (checking[j][z])
      h = max(h,j);                  #        add(j,conf_set[z]);
                                     #    h = max(conf_set[z]);
                                     #    merge(conf_set[h],conf_set[z]);
  for (i = z; i >= h; i--)           =    for (i = z; i >= h; i--)
    restore(i);                      =      restore(i);
  return(h);                         =    return(h);
}                                    =  }
```

In addition to the data structures inherited from FC, the FC hybrids use the data structures of the backward checking algorithms. FC-BJ employs the array *max_check* of BJ, whereas FC-CBJ uses *conf_set* of CBJ. Functions $consistent(i)$ and $restore(i)$ are identical to those in FC.

The FC hybrids attempt to combine the advantages of forward checking and backjumping. However, the resulting algorithms are complex and hard to understand in detail.

Chapter 3

Four Basic Algorithms

In this chapter we formally analyze the behaviour and prove the correctness of four well-known backtracking algorithms: Chronological Backtracking (BT), Backjumping (BJ), Conflict-Directed Backjumping (CBJ), and Forward Checking (FC). The chapter is organized as follows. Section 3.1 shows how the algorithms work on a nontrivial example. Section 3.2 defines backjumps in terms of inconsistency between variables and instantiations. Section 3.3 points out the modifications which have to be introduced in order for the algorithms to find all solutions. Section 3.4 contains the fundamental basic theorems describing the behaviour of the backtracking algorithms. Section 3.5 presents the hierarchy with respect to the number of visited nodes. Section 3.6 contains correctness proofs.

3.1 A Few Insights

Let us start by presenting an example which illustrates the differences between these four algorithms.

Example 3. Figure 3.3 shows a fragment of the backtrack tree generated by BT for the 6-queens problem. White dots denote consistent nodes. Black dots denote inconsistent nodes. The dark-shaded part of the tree denotes two nodes which are skipped by BJ. The light-shaded part denotes nodes which are skipped by CBJ. The numbered consistent nodes are the nodes visited by FC. Dashed arrows represent backjumps. The left one is performed by CBJ, and the right one is performed by BJ. Chronological backtracks are not represented. 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 which have been excluded due to the already placed queens. The numbers inside the excluded squares indicate the earliest placed queen responsible for their exclusion; 1,2,3 correspond to the first, second, and third queen respectively.

The search performed by BT on the subtree rooted at node 253 is uneventful.

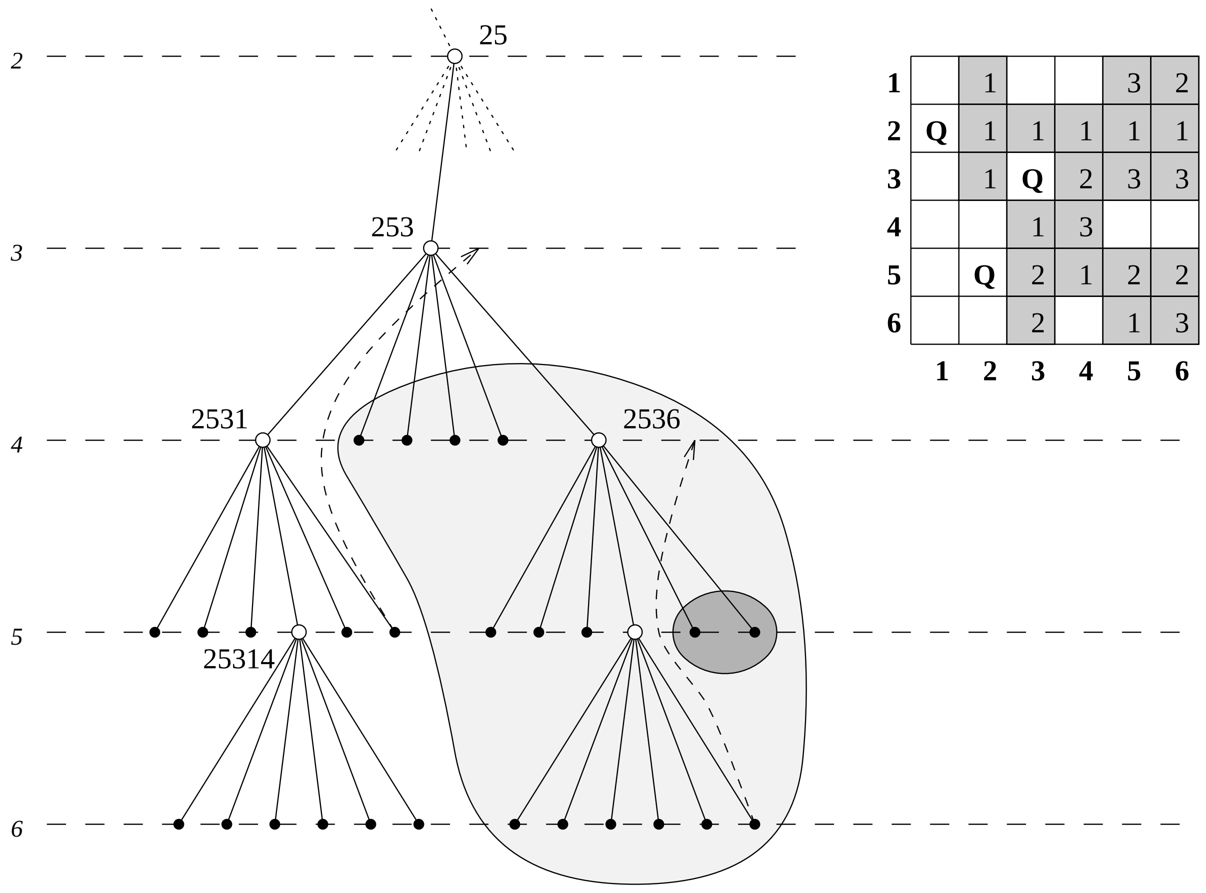


Figure 3.3: A fragment of the BT backtrack tree for the 6-queens problem.

Every consistent node is fully expanded. Two dead-ends are encountered, and a total of 31 nodes are visited.

BJ manages to skip two nodes in the subtree. The algorithm detects a dead-end at variable x_6 when it tries to expand node 25364. It then backtracks to the highest variable in conflict with x_6, in this case x_4. 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 inconsistency. Nodes 25365 and 25366 may be safely skipped.

Note that backtracking to level i does not mean that the next visited node will be on the level i. In our example, BJ after backjumping from node 253646 to level 4 finds that there are no more values to be tried for variable x_4; therefore, it chronologically backtracks to x_3 and visits node 254.

CBJ achieves considerable savings as it skips seventeen nodes in the subtree. 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 contains numbers $1, 2, 3$, and 5 as the reasons for the unavailability of the squares. 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 is now $\{1, 2, 3\}$. After trying the two remaining values of x_5, CBJ backjumps to x_3 skipping the rest of the subtree. 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 3.3 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 either x_5 or x_6 taken separately.

Figure 3.5 shows a detailed trace of CBJ on a larger subtree rooted at node 25. The four columns in the lower part of Figure 3.5 correspond to the subtree shown in Figure 3.3. Straight solid arrows represent node expansion. The squares marked with 'x' are the ones that are not expanded by CBJ because of the backjump from x_5 to x_3. Dashed arrow represent backtracks, but this time both chronological and non-chronological backtracks are shown. The conflict sets, which are passed backwards, are shown along the backtracks (the values of d should be ignored for the time being). The reader may want to read the previous paragraph again, this time with Figure 3.5 in front of him. A good starting point is the dead-end in the lower right corner of the figure, which corresponds to nodes 253141–253146.

FC, in contrast with the backward checking algorithms, visits only consistent nodes, in this case 253, 2531, 25314, and 2536. The board in Figure 3.3 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.

3.2 Backjump Lemmas

Let us now formalize the intuition about backjumps in the form of two lemmas. The lemmas will later enable us to prove theorems about the backtrack trees of BJ and CBJ.

In both lemmas we use C_i to denote a tuple composed of instantiations of selected past variables. This is the only time in this thesis when we use tuples which are not composed of consecutive instantiations and so cannot be identified with search tree nodes. Figure 3.4 shows one such tuple in the regular 6-queens problem. Tuple

$((x_2,5),(x_4,6),(x_5,3))$ is inconsistent with variable x_6. All nodes that contain this tuple, e.g. $((x_1,1),(x_2,5),(x_3,2),(x_4,6),(x_5,3))$, are also inconsistent with x_6.

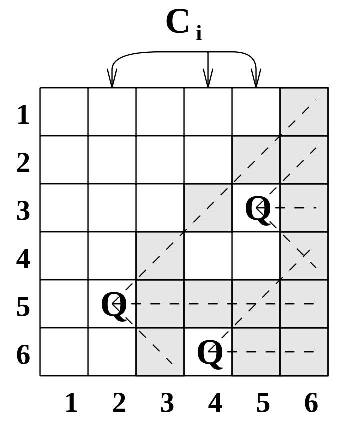


Figure 3.4: A tuple composed of non-consecutive instantiations in the 6-queens problem.

Lemma 1 *If BJ backtracks to variable x_h from a dead-end at variable x_i then $(X_1,\ldots,X_h)$ is inconsistent with x_i.*

Proof. A dead-end happens when all values of the current variable are rejected by BJ. For each rejected value we can name the past variable responsible for the rejection: it is the variable against which the particular consistency check failed. These variables are said to be in conflict with the current variable. 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 which are in conflict with x_i. Clearly, C_i is inconsistent with x_i. As C_i consists of instantiations of past variables only, it is a subtuple of $(X_1,\ldots,X_{i-1})$. Moreover, since X_h is the instantiation of the highest variable in C_i, C_i is a subtuple of $(X_1,\ldots,X_h)$. Therefore, $(X_1,\ldots,X_h)$ is also inconsistent with x_i. □

In order to prove the next lemma, we need the notion of *backtrack depth.* Informally, the depth of a backtrack is the distance, measured in backtracks, from the backtrack destination to the "farthest" dead-end. The definition is recursive:

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

Figure 3.5 contains six backtracks. Three of them are performed from dead-ends, and so they are of depth $d=1$. The other three backtracks are of depth $d>1$ because they are performed from variables which are the destinations of other backtracks.

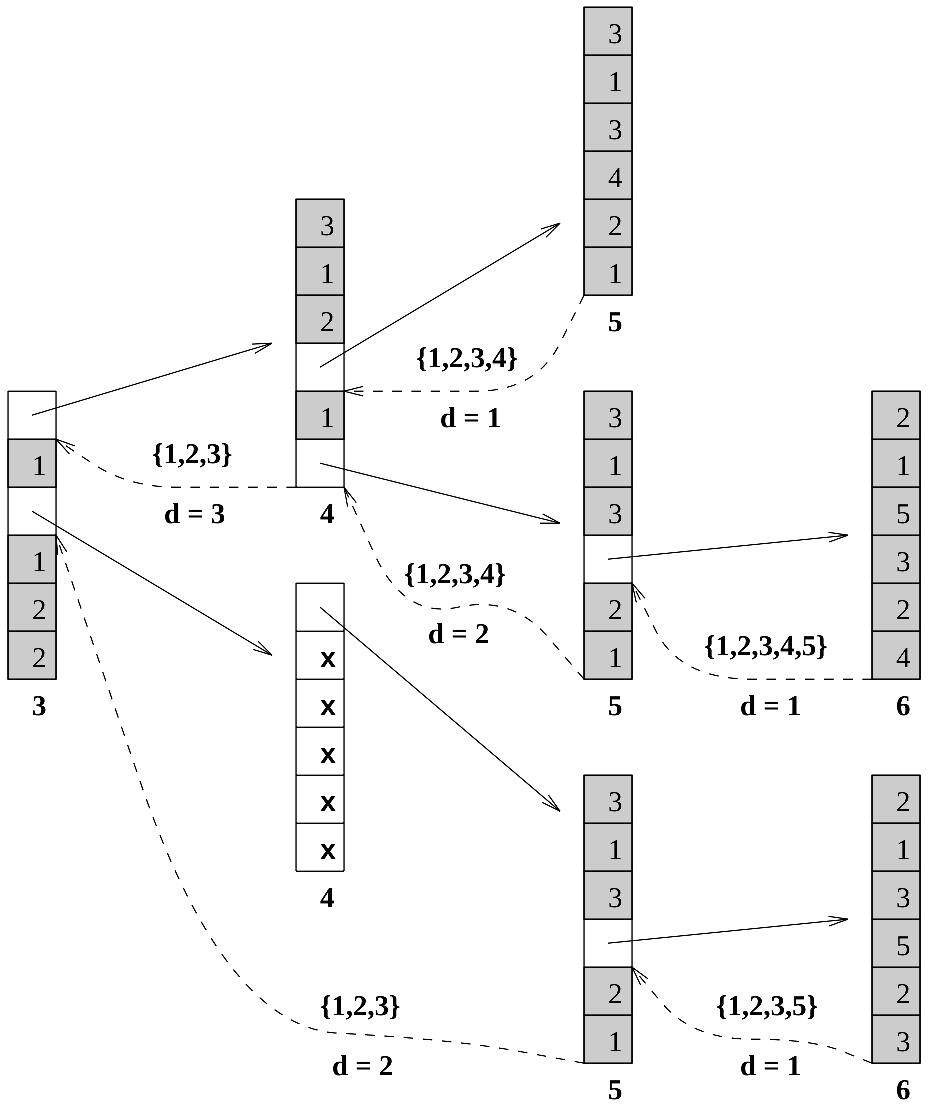


Figure 3.5: CBJ's backtracking behaviour on the subtree rooted at node 25.

Lemma 2 is given under the assumption that CBJ terminates after finding the first solution. Later, we change this assumption to a weaker one.

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* [1].

Proof. The variable x_h, which is the highest variable in the conflict set of x_i, is chosen by CBJ as the point of backtrack from x_i. The conflict set of x_i is the union

[1]More precisely, S is the set of variables that contributed their conflict sets to the conflict set of x_i, but we will not use this fact in the thesis.

of the set of all past variables in conflict with x_i and all conflict sets inherited from variables higher than x_i.

The proof proceeds by induction on the depth of backtrack. For the basis, consider a backtrack of depth 1, that is, one performed from a dead-end. Since no conflict sets are inherited from higher variables, the conflict set of x_i contains only variables in conflict with x_i. Clearly, C_i 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 depth less than d and consider a backtrack of depth d. Let C_i^t denote the tuple produced by extending C_i with some instantiation $t \in D_i$. C_i^t may be consistent or not[2]. If C_i^t is consistent, there must have been a backtrack of depth less than d from some variable x^t to variable x_i. From the inductive hypothesis we know that the tuple 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 the elements of the conflict set of x^t, except x_i, C_i^t is also inconsistent with S^t. If, on the other hand, C_i^t is inconsistent itself, it is also inconsistent with any set of variables, so take $S^t = \emptyset$. Therefore, for every instantiation $u \in D_i$, C_i^u is inconsistent with the set comprising all S^t sets, namely $\bigcup_{t \in D_i} S^t$. This in turn implies that C_i is inconsistent[3] with the set $S = \{x_i\} \cup \bigcup_{t \in D_i} S^t$. $\square$

3.3 Finding All Solutions

Faced with a constraint satisfaction problem we can ask several different questions about it:

- Is there a solution?
- How many solutions are there?
- What is the solution?
- What are all solutions?

Nudel [16] distinguishes four variations of the consistent labelling problem, which correspond to the four above questions respectively:

- Consistent Labelling Decision Problem (CLDP)
- Consistent Labelling Enumeration Problem (CLEP)
- Consistent Labelling Search Problem (CLSP)

[2]For example, take the backtrack from x_4 to x_3 in Figure 3.5. $C_4 = ((x_1,2),(x_2,5),(x_3,1))$. If we take $t = (x_4,6)$, we get $C_4^t = ((x_1,2),(x_2,5),(x_3,1),(x_4,6))$, which is consistent itself, but inconsistent with $S^t = \{x_5,x_6\}$. If we take $t = (x_4,5)$, we get $C_4^t = ((x_1,2),(x_2,5),(x_3,1),(x_4,5))$, which is inconsistent itself, and so also inconsistent with $\emptyset$.

[3]In our example, $S = \{x_4\} \cup (\emptyset \cup \emptyset \cup \emptyset \cup \{x_5\} \cup \emptyset \cup \{x_5,x_6\}) = \{x_4,x_5,x_6\}$.

- Consistent Labelling Generation Problem (CLGP)

The last variant is the most general since it comprises all others. In this work, we are interested mainly in algorithms which find all solutions. However, backtracking algorithms are usually designed to stop after finding the first solution, and have to be modified in order to solve CLGP. A simple change to the termination condition is sufficient for some algorithms (e.g.. BT, BJ, FC), but in the case of CBJ and its hybrids further modifications are necessary.

Example 4. The confused n-queens problem, described in [14], is how to place n queens on a $n \times n$ chess board, one queen per column, so that every queen attacks every other queen. Suppose we change the *solution()* function of CLSP versions so that it does not terminate the algorithm, and then use BJ and CBJ to solve the confused 3-queens problem. BJ correctly generates all solutions; CBJ, however, misses three of nine solutions. Figure 3.6 shows one of the solutions detected by both algorithms, namely (2,1,2). At this moment, the conflict set of x_3 contains only one variable, x_1, which causes CBJ to backtrack directly to x_1. Two subsequent solutions, (2,2,2) and (2,3,2), are thus pruned out.

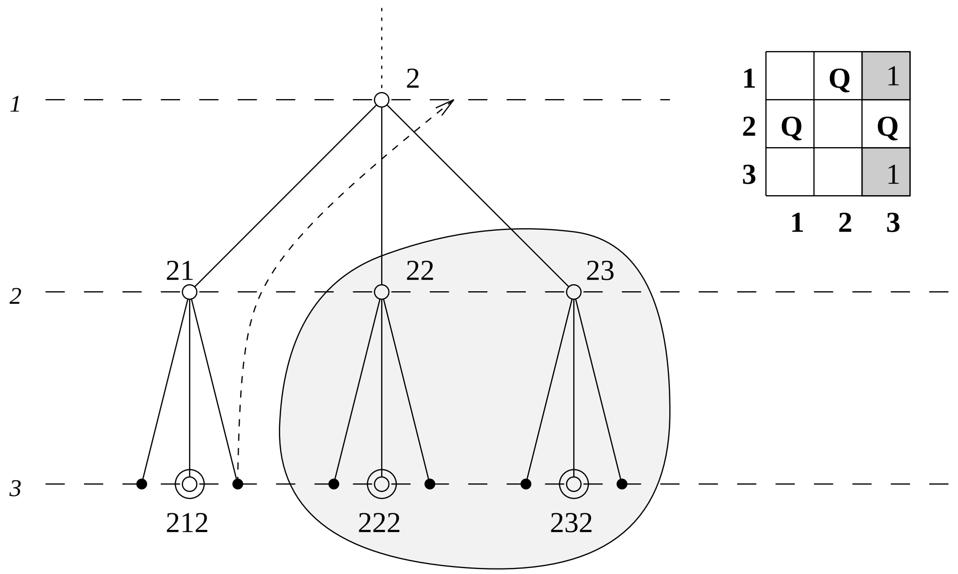


Figure 3.6: A fragment of backtrack tree for the confused 3-queens problem.

The problem here is that the conflict sets of CBJ are meant to indicate which instantiations are responsible for some previously discovered inconsistency. However, after a solution is found, conflict sets cannot always 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 causes of CBJ backtracks: (1) detecting an inconsistency, and (2) searching for other solutions. In the latter case the backtrack must be always chronological, that is, to the immediately preceding variable (otherwise we risk pruning out some of the solutions).

Although it is possible to make CBJ find all solutions without adding new data structures, we decided to adopt the following approach for its conceptual clarity. The modified CBJ employs a one-dimensional boolean array of size n, called *cbf* (chronological backtrack flag). When set, an array entry signals that the corresponding conflict set no longer has the intended meaning. Every time a new variable is chosen for instantiation, the corresponding *cbf* entry is set to zero. After every discovered solution, all entries in the *cbf* array are set to one. *cbf* is used when there are no remaining values to be tried for the current variable. If the corresponding *cbf* entry is set, the backtracking point is the variable immediately preceding the current one in the instantiation order. Otherwise, the highest variable in the conflict set of the current variable is chosen.

Lemma 2 as formulated in the previous chapter does not hold for every backtrack in the modified CBJ. Indeed, the definition of backtrack depth does not apply to backtracks caused by searching for other solutions. However, if we restrict ourselves to the backtracks performed when the corresponding *cbf* entry is zero, the lemma and the proof are still valid. Since the backtracks performed when the *cbf* flag is set are always chronological and do not involve node skipping, the lemma holds for all backjumps performed by CBJ. Therefore, we can be sure that whenever nodes are skipped by backjumps, it is because of some previously detected inconsistency. We use this fact in the proofs of the theorems presented in the following section.

Not every backtracking algorithm has to be modified in this way to find all solutions. BT and FC never backjump. BJ does backjumps, but only from dead-ends, and there is no dead-end when a solution is found. In the case of these three algorithms we need only to change the termination condition to obtain CLGP versions.

3.4 Fundamental Theorems

We now present several theorems which describe the behaviour of four basic backtracking algorithms: BT, BJ, CBJ, and FC. 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 first with the more general problem of finding all solutions. Then, we point out which of the results are valid when only one solution is sought.

3.4.1 Characterizing Conditions for BT, BJ, and CBJ

The following three conditions are helpful in characterizing nodes in the search tree:

(1) A node's parent is consistent.

(2) A node's parent is consistent with all variables.

(3) A node's parent is consistent with all sets of variables.

Note that Conditions 2 and 3 are not equivalent. Condition 2 states that for every individual variable, there exists an instantiation which is consistent with a certain tuple. Condition 3 states that in addition all instantiations of the individual variables must be consistent with one another. A tuple that satisfies Condition 3 is in fact a part of a full solution. If there is no solution to the network then no tuples satisfy Condition 3. Naturally, Condition 3 implies Condition 2, which in turn implies Condition 1. Interestingly, we can use the three conditions to specify the sufficient conditions for nodes to be visited by the three backward checking algorithms. The following theorems formalize this observation.

Theorem 1 *BT visits a node if its parent is consistent.*

Proof. Suppose that node $(X_1, \ldots, X_{i-1})$ is consistent, and its child $p = (X_1, ..., X_i)$ is not visited by BT. Take the highest j, $j \leq i - 1$, such that node $p' = (X_1, \ldots, X_j)$ is visited by BT. Node p' is a proper ancestor of node p and is consistent because $(X_1, \ldots, X_j)$ is a subtuple (not necessarily proper) of tuple $(X_1, \ldots, X_{i-1})$. When BT visits p', all consistency checks between X_j and previous instantiations succeed. The branch is extended by instantiating the next variable x_{j+1} to *each* of the values in its domain, including X_{j+1}. The node $(X_1, \ldots, X_j, X_{j+1})$ is thus visited by BT, a contradiction. $\Box$

Theorem 2 *BJ visits a node if its parent is consistent with all variables.*

Proof. Suppose that node $(X_1, \ldots, X_{i-1})$ is consistent with all variables, and its child $p = (X_1, \ldots, X_i)$ is not visited by BJ. Take the highest j, $j \leq i - 1$, such that node $p' = (X_1, \ldots, X_j)$ is visited by BJ. Node p' is a proper ancestor of node p and is consistent with all variables because $(X_1, \ldots, X_j)$ is a subtuple (not necessarily proper) of tuple $(X_1, \ldots, X_{i-1})$. When BJ is at node p', all consistency checks between X_j and previous instantiations succeed. The only reason for not instantiating the next variable x_{j+1} to X_{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 $(X_1, \ldots, X_g)$ is inconsistent with x_h, which contradicts the initial assumption that node $(X_1, \ldots, X_{i-1})$ is consistent with all variables. $\Box$

Theorem 3 *CBJ visits a node if its parent is consistent with all sets of variables.*

Proof. Suppose that node $(X_1, \ldots, X_{i-1})$ is consistent with all sets of variables, and its child $p = (X_1, \ldots, X_i)$ is not visited by CBJ. Take the highest j, $j \leq i-1$, such that node $p' = (X_1, \ldots, X_j)$ is visited by CBJ. Node p' is a proper ancestor of node p and is consistent with all sets of variables because $(X_1, \ldots, X_j)$ is a subtuple (not necessarily proper) of tuple $(X_1, \ldots, X_{i-1})$. When CBJ is at node p', all consistency checks between X_j and previous instantiations succeed. The only reason for not instantiating the next variable x_{j+1} to X_{j+1} can be a backjump from some variable x_h to some variable x_g, where $g \leq j$ and $h \geq j + 2$. From Lemma 2 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 $(X_1, \ldots, X_{i-1})$ is consistent with all sets of variables. □

The above theorems allow us to identify only the nodes which are necessarily visited by the algorithms. However, these are not *all* nodes that the algorithms visit. For example, in Figure 3.3 BJ visits the node 25364 even though it's parent is inconsistent with x_6. This happens because backward checking algorithms "look backward" when they search for a solution. Unlike FC, they have to actually "hit" an inconsistency in order to discover it. We would like to be able to specify a class of nodes which are guaranteed *not* to be visited by algorithms. In other words, we would like to have not only the sufficient, but also the necessary condition for a node to be visited. Ideally, the sufficient and the necessary conditions should be the same.

The following three theorems formalize a trivial observation that the backward checking algorithms expand only consistent nodes. Since all three algorithms BT, BJ, and CBJ perform consistency checking in the same way, there is no need for more than one proof.

Theorem 4 *BT visits a node only if its parent is consistent.*

Proof. Suppose that in the search tree there exists a node $p = (X_1, \ldots, X_i)$ which is visited by BT, and at the same time its parent $(X_1, \ldots, X_{i-1})$ is inconsistent. Take the highest j, $j \leq i-1$, such that node $(X_1, \ldots, X_{j-1})$ is consistent. It is guaranteed to exist because all first level nodes are consistent by the assumption that the constraint network is already node consistent. Node $p' = (X_1, \ldots, X_j)$ is a proper ancestor of node p; and so p' is also visited. When BT visits p', a consistency check fails between X_j and a previous instantiation, thus causing the branch to be abandoned. Therefore, no descendants of p' are visited by BT, a contradiction. □

Theorem 5 *BJ visits a node only if its parent is consistent.*

Proof. The same as for Theorem 4. □

Theorem 6 *CBJ visits a node only if its parent is consistent.*

Proof. The same as for Theorem 4. □

3.4.2 Characterizing Conditions for FC

For FC we can give a more precise characterization of the set of visited nodes. The condition used in the following theorem is referred to as Condition 4.

(4) A node is consistent and its parent is consistent with all variables.

Note that the condition consist of two conjuncts, the second of which is identical to Condition 2. Therefore, Condition 4 implies Condition 2. It does not, however, imply Condition 3.

Theorem 7 *FC visits a node if it is consistent and its parent is consistent with all variables.*

Proof. Suppose that node $(X_1, \ldots, X_{i-1})$ is consistent with all variables, and its child $p = (X_1, \ldots, X_i)$ is consistent, but not visited by FC. Take the highest j, $j \leq i-1$, such that node $p' = (X_1, \ldots, X_j)$ is visited by FC, but its child $(X_1, \ldots, X_j, X_{j+1})$ is not visited by FC. Node p' is a proper ancestor of node p and is consistent with all variables because $(X_1, \ldots, X_j)$ is a subtuple (not necessarily proper) of tuple $(X_1, \ldots, X_{i-1})$. When FC visits node p', none of the domains of the future variables is annihilated. The branch is therefore extended by instantiating the next variable x_{j+1} to *each* of the values in its filtered domain. Since tuple $(X_1, \ldots, X_i)$ is consistent, its subtuple $(X_1, \ldots, X_{j+1})$ is also consistent, so the filtered domain of x_{j+1} must still contain X_{j+1}. Node $(X_1, \ldots, X_j, X_{j+1})$ is thus visited by FC, a contradiction. $\Box$

The necessary condition for a node to be visited by FC is identical to the sufficient condition.

Theorem 8 *FC visits a node only if it is consistent and its parent is consistent with all variables.*

Proof. We prove the second conjunct first. Suppose that FC visits node $p = (X_1, \ldots, X_i)$ although its parent $(X_1, \ldots, X_{i-1})$ is inconsistent with some variable. Take the highest j, $j \leq i-1$, such that node $(X_1, \ldots, X_{j-1})$ is consistent with all variables. Node $p' = (X_1, \ldots, X_j)$ 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 $p = (X_1, \ldots, X_i)$ although p is inconsistent. Take the lowest k, $k \leq i-1$, such that instantiation X_k is inconsistent with instantiation X_i. When FC is at node $(X_1, \ldots, X_k)$, the value X_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. $\Box$

3.4.3 Summary of Conditions

Figure 3.7 summarizes the results of this section. The numbers denote that a node satisfies a given condition. The names of algorithms denote that a node is visited by a given algorithm. The arrows represent implications, and are annotated by the numbers of the corresponding theorems. For example, Theorem 7, which says that a node is visited by FC if it satisfies Condition 4, is represented by the arrow from '4' to 'FC'.

There is a significant difference between the algorithms that use chronological backtracking and the algorithms that use backjumping. BT and FC are completely characterized; that is, for every node we can decide whether it is visited by the algorithm without having to generate the whole backtrack tree. In the case of BJ and CBJ, however, there is a set of nodes for which we are unable to tell if they belong to the algorithm's search tree or not. The above theorems are not powerful enough for this purpose. It is an open question if better characterizing conditions can be found.

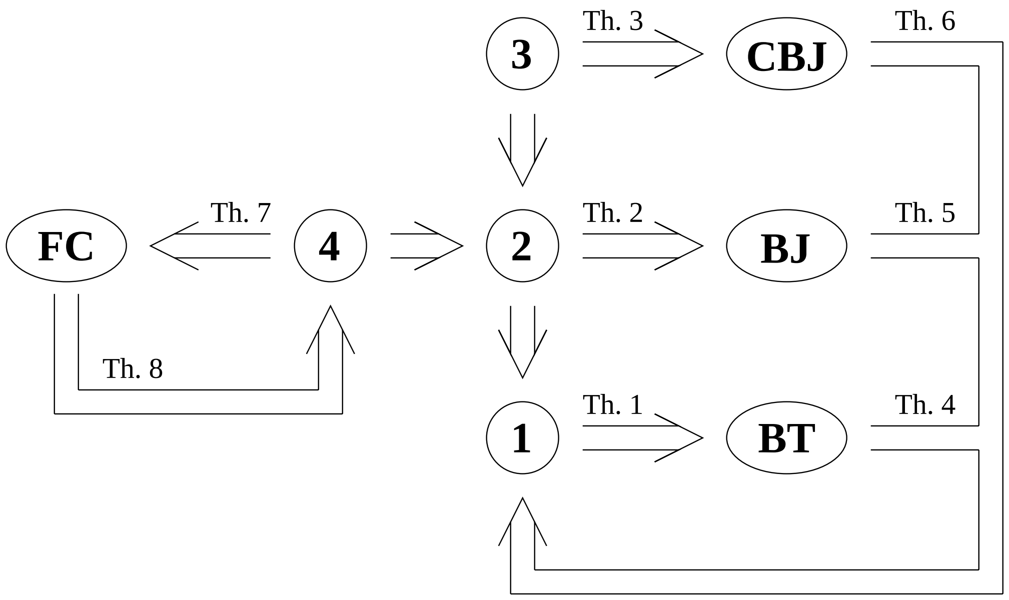


Figure 3.7: Conditions graph.

For CLSP versions (one solution sought), the "only if" theorems (4, 5, 6, and 8) are still valid, since in the respective proofs we do not use the assumption that the search is continued until all possibilities are exhausted. However, the proofs of the "if" theorems (1, 2, 3, and 7) are based on the assumption that a node is always extended to *each* value in the domain of the next variable. This is true only if the search is not interrupted until all possibilities are exhausted. Therefore, the "if" theorems are valid for CLSP only if we restrict ourselves to the search tree nodes preceding (in the preorder traversal) the last node visited by a backtracking algorithm.

3.5 Hierarchy

It turns out that the nodes visited by the algorithms often form sets which include one another. Let us look at Figure 3.7 again. If there is a path along the arrows leading from algorithm $\mathcal{A}$ to algorithm $\mathcal{B}$ then we can conclude that $\mathcal{B}$ visits all nodes that $\mathcal{A}$ visits. This observation immediately yields the following four corollaries. Note that these corollaries are valid regardless of the constraint satisfaction problem being solved.

Corollary 1 *BT visits all nodes that BJ visits.*

Proof. The corollary states that if a node is visited by BJ then it is also visited by BT. From Theorem 5 we know that all nodes visited by BJ have parents which are consistent. Theorem 1 guarantees that all such nodes are visited by BT. Therefore, if a node is visited by BJ, it is also visited by BT. □

Corollary 2 *BT visits all nodes that CBJ visits.*

Proof. From Theorem 6 we know that all nodes visited by CBJ have parents which are consistent. Theorem 1 guarantees that all such nodes are visited by BT. Therefore, if a node is visited by CBJ, it is also visited by BT. □

Corollary 3 *BT visits all nodes that FC visits.*

Proof. From Theorem 8 we know that all nodes visited by FC have parents which are consistent with all variables (and so the parents are themselves consistent). Theorem 1 guarantees that all such nodes are visited by BT. Therefore, if a node is visited by FC, it is also visited by BT. □

The next corollary is the most interesting. The relationship between BJ and FC has never been stated before, although the two algorithms have been often empirically compared. This is probably due to their apparent dissimilarity.

Corollary 4 *BJ visits all nodes that FC visits.*

Proof. From Theorem 8 we know that all nodes visited by FC have parents which are consistent with all variables. Theorem 2 guarantees that all such nodes are visited by BJ. Therefore, if a node is visited by FC, it is also visited by BJ. □

The relationship between BJ and CBJ, although not implied by the theorems, can also be proven using the two lemmas from Section 3.2. The proof is rather technical and may be skipped without affecting the understanding of the rest of the thesis.

Theorem 9 *BJ visits all nodes that CBJ visits.*

Proof. Suppose that in the search tree of CBJ there is a node p at level h which is not visited by BJ (Figure 3.8, left). The only reason for skipping p can be a backjump performed by BJ from some node q at level k to level $g < h$. Recall that BJ performs backjumps only after detecting a dead-end, and that in such a case it behaves exactly like CBJ. Therefore, node q could not 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 at level j to level $i < k$ (Figure 3.8, right).

The nodes in question are
$p = (X_1, \ldots, X_h)$,
$q = (Y_1, \ldots, Y_k)$,
$r = (Z_1, \ldots, Z_j)$.
From Lemma 1 we have:

$$\neg\ ((Y_1, \ldots, Y_g)\ \Delta\ x_k).$$

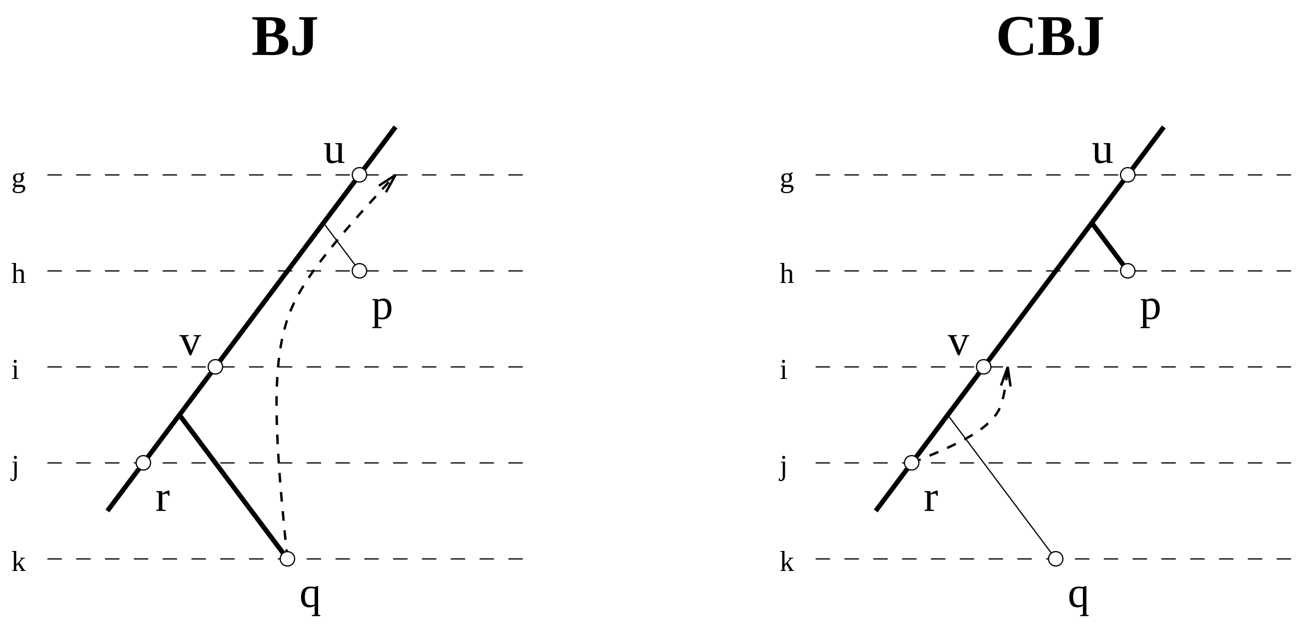


Figure 3.8: A hypothetical situation when CBJ visits a node not visited by BJ.

From Lemma 2 we have:

$$\neg\ ((Z_1, \ldots, Z_i)\ \Delta\ S)$$

where $S \subseteq \{x_j, \ldots, x_n\}$.
From the properties of trees we have:
$u = (X_1, \ldots, X_g) = (Y_1, \ldots, Y_g) = (Z_1, \ldots, Z_g)$,
$v = (Y_1, \ldots, Y_i) = (Z_1, \ldots, Z_i)$.
Therefore, also

$$\neg\ ((Y_1, \ldots, Y_i)\ \Delta\ S),$$

and

$$\neg\ ((Z_1, \ldots, Z_g)\ \Delta\ x_k).$$

Let us denote the highest 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. □

The above four corollaries and one theorem enable us to construct a partial order of backtracking algorithms with respect to the number of visited nodes. Figure 3.9

shows the hierarchy for the four basic algorithms analyzed so far. BT generates the biggest backtrack tree, which contains all nodes that the other algorithms visit on the same problem. BJ visits more nodes than CBJ or FC. The order would be linear if there was a relationship between FC and CBJ, but this not the case. Figure 3.3 provides a counterexample: some nodes visited by CBJ are not visited by FC, and vice versa.

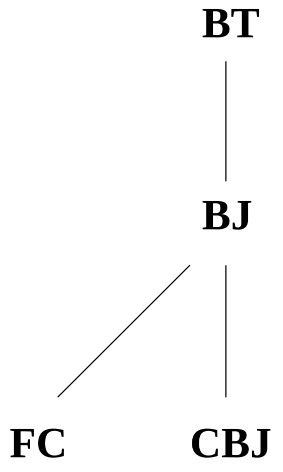


Figure 3.9: The hierarchy with respect to the number of visited nodes.

3.6 Correctness

It is surprisingly difficult to find the correctness proofs of most backtracking algorithms. BT and FC, being conceptually simple, probably do not require rigorous proofs. It is not immediately clear, however, that BJ and CBJ are correct.

Ginsberg [8] presents five algorithms using a new notation. Ginsberg's Algorithm 2.5, which is referred to as depth-first search, is most probably equivalent to BT. It is not clear, however, if Algorithm 3.3, which Ginsberg calls Backjumping, is equivalent to BJ or CBJ, or is a completely new algorithm. Two propositions are of interest to us:

> **Proposition 2.7** [8] *Algorithm 2.5 is equivalent to depth-first search and therefore complete.*
> **Proposition 3.4** [8] *Backjumping is complete and always expands fewer nodes than does depth-first search.*

The completeness proofs are difficult to follow. That Backjumping always expands fewer nodes than depth-first search is not proven. It should be noted that it is not difficult to find problems on which BJ and CBJ expand the same number of nodes as BT.

Prosser in [20] mentions there exists an informal correctness proof of CBJ by Tsang. As it has never been published, nothing more can be said about it.

The correctness of the four basic algorithms is almost immediate from the theorems given in Section 3.4. For each algorithm we prove that it is sound (finds only solutions) and complete (finds all solutions). That all the algorithms terminate is clear.

Corollary 5 *BT is correct.*

Proof (soundness). A solution is claimed by BT if all consistency checks succeed at an n-level node. It means that $(X_1, \ldots, X_n)$ is visited and $\forall i < n : X_i$ is consistent with X_n. Theorem 4 implies that its parent $(X_1, \ldots, X_{n-1})$ is consistent. Therefore, $(X_1, \ldots, X_n)$ is consistent.

Proof (completeness). Suppose that some n-level node $(X_1, \ldots, X_n)$ in the search tree is consistent. Then, its parent $(X_1, \ldots, X_{n-1})$ is consistent as well. From Theorem 1 we know that $(X_1, \ldots, X_n)$ is visited by BT. Since all consistency checks between X_n and previous instantiations must succeed, a solution is claimed by BT. □

Corollary 6 *BJ is correct.*

Proof (soundness). The same as the proof of the soundness of BT, except that we use Theorem 5.

Proof (completeness). Suppose that some n-level node $(X_1, \ldots, X_n)$ in the search tree is consistent. Then, its parent $(X_1, \ldots, X_{n-1})$ is consistent as well, and it is also consistent with x_n. Therefore, $(X_1, \ldots, X_{n-1})$ is consistent with all variables. From Theorem 2 we know that $(X_1, \ldots, X_n)$ is visited by BJ. Since all consistency checks between X_n and previous instantiations must succeed, a solution is claimed by BJ. □

Corollary 7 *CBJ is correct.*

Proof (soundness). The same as the proof of the soundness of BT, except that we use Theorem 6.

Proof (completeness). Suppose that some n-level node $(X_1, \ldots, X_n)$ in the search tree is consistent. Then, its parent $(X_1, \ldots, X_{n-1})$ is consistent as well, and it is also consistent with the set $\{x_n\}$. Therefore, $(X_1, \ldots, X_{n-1})$ is consistent with all sets of variables. From Theorem 3 we know that $(X_1, \ldots, X_n)$ is visited by CBJ. Since all consistency checks between X_n and previous instantiations must succeed, a solution is claimed by CBJ. □

Corollary 8 *FC is correct.*

Proof (soundness). A solution is claimed by FC if an n-level node $p = (X_1, \ldots, X_n)$ is reached. Since Theorem 8 guarantees that a node visited by FC is consistent, p must be consistent.

Proof (completeness). Suppose that some n-level node $(X_1, \ldots, X_n)$ in the search tree is consistent. Then, its parent $(X_1, \ldots, X_{n-1})$ is consistent as well, and it is also consistent with $\{x_n\}$. Therefore, $(X_1, \ldots, X_{n-1})$ is consistent with all variables. From

Theorem 7 we know that $(X_1, \ldots, X_n)$ is visited by FC. Since this is a n-level node, a solution is claimed by FC. $\Box$

The above theorems prove the correctness of the CLGP versions of the algorithms. For BT, BJ, and FC, the correctness of the standard CLSP versions is obvious: if an algorithm is guaranteed to correctly find all solutions then it also correctly finds the first solution. It is only a little bit more complicated for CBJ because the standard CLSP version of the algorithm does not use *cbf* array. However, since none of the *cbf* entries is set before finding the first solution, the additional array does not influence the behaviour of the CLGP version in this phase of the search. Therefore, we can conclude that the standard CLSP version of CBJ is also correct.

Chapter 4

The Six Remaining Algorithms

In this chapter we analyze somewhat less formally the six remaining backtracking algorithms described in chapter 2. No theorems or proofs are given. All our claims are conjectures based on empirical tests and a careful analysis of the algorithms.

In the first three sections we discuss three groups of algorithms:

- Backmarking and its hybrids.
- Graph-Based Backjumping.
- Forward Checking hybrids.

In the last section we present the final hierarchies of all backtracking algorithms analyzed in this thesis.

4.1 Backmarking and its Hybrids

One thing that Backmarking (BM), Backmarking and Backjumping (BMJ), and Backmarking and Conflict-Directed Backjumping (BM-CBJ) have in common is that they use a backmarking scheme. A backmarking scheme 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. In this section, we first thoroughly analyze the behaviour of the backmarking algorithms on a small example, and then propose a modification to BMJ.

4.1.1 The Problem With BMJ

BMJ is a synthesis of BM and BJ; the hybrid, however, does not retain all the power of each base algorithm in terms of consistency checks. Prosser [17] 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. The purpose of the following example is to explain why it happens.

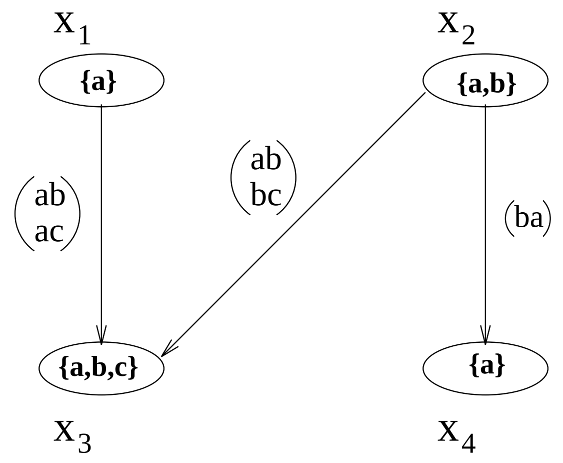


Figure 4.10: The constraint network of Example 5.

Example 5. Consider the constraint network of four variables represented by the graph in Figure 4.10. 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 natural order. It is easy to verify that there is only one solution to the network.

Figure 4.11 shows the backtrack tree generated by BT. BT visits 11 nodes, and performs 17 consistency checks, an improvement over the naive "generate and test" approach which involves 23 nodes.

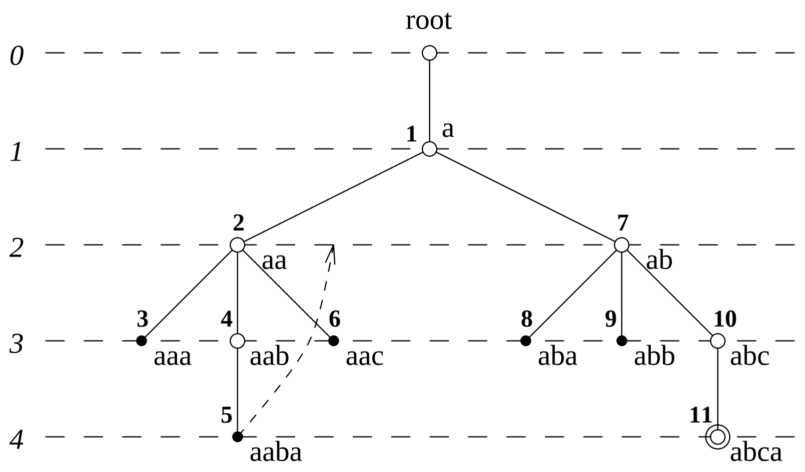
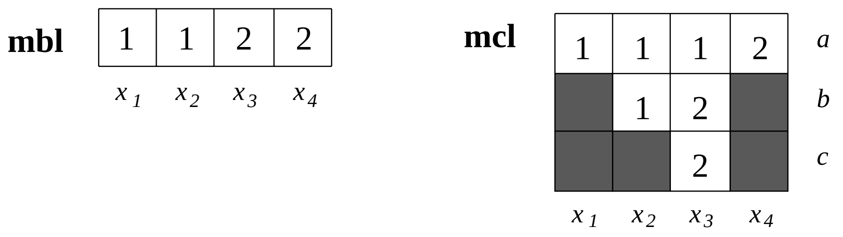


Figure 4.11: The backtrack tree generated by BT on the constraint network of Example 5.

Let r be the number of consistency checks that BT performs at a given node. If the node is inconsistent, r is the number of the lowest variable whose instantiation is inconsistent with the instantiation of the current variable x_i. If the node is consistent, all checks succeed, and so $r = i - 1$. We will use r to compute the consistency checks savings achieved by the backmarking algorithms.

We already know from the previous chapter that the set of nodes visited by BJ is a subset of the set of nodes visited by BT. In this small example BJ manages to perform only one backjump, which is represented in Figure 4.11 by the dashed arrow. When node 5 is visited, the entry *max_check*[4] is set to 2 because variable x_4 is inconsistent with the instantiation (x_2, a). By jumping back to x_2, BJ skips one node and saves two consistency checks. Following a general rule, at every other node BJ performs the same number of checks as BT.

In the case of BM the opposite is true. BM visits exactly the same nodes as BT but at some of them performs less consistency checks. In our example BM behaves like BT when it explores the subtree rooted at node 2 (the left subtree). However, during the search, information about consistency checks is accumulated in its data structures *mbl* and *mcl*. This information is utilized when BM visits the subtree rooted at node 7 (the right subtree). The contents of the arrays just before BM visits node 7 are shown in Figure 4.12. For the expository purposes, the arrays are transposed.



mbl

x_1	x_2	x_3	x_4
1	1	2	2

mcl

	x_1	x_2	x_3	x_4
a	1	1	1	2
b	▨	1	2	▨
c	▨	▨	2	▨

Figure 4.12: Arrays *mbl* and *mcl* of BM before node 7 is visited.

Let us denote by |mcl| and |mbl| respectively the values of the *mcl* and *mbl* entries that are consulted when a given node is visited. If |mcl| is smaller than |mbl|, no consistency checks are performed by BM (type-A savings as described in section 2.2.6). This is because the instantiation which causes the node to be inconsistent has not been changed since the *mcl* entry was last updated. The number of saved checks is thus equal to $r =$ |mcl|. If |mcl| is greater than or equal to |mbl|, only those instantiations which have changed are checked (type-B savings). The instantiations of variables lower than |mbl| are guaranteed to succeed, and so the number of saved checks is equal to |mbl| -1. Therefore, the savings made at each node can be given by a simple formula: min(|mcl|,|mbl| -1).

BMJ attempts to combine node skipping with consistency check saving. Its backtrack tree is always the same as the backtrack tree of BJ. In our example, it saves two checks when it backjumps over node 6, but on the right subtree it performs *three* checks more than BM. On the whole network BMJ performs more checks than BM.

To see why this happens, consider node 9, which corresponds to the tuple $((x_1, a),(x_2, b),(x_3, b))$. The instantiation of x_1 has not changed since the consistency check between instantiations (x_1, a) and (x_3, b) was performed at node 4. According to the definition, *mbl*[3] should contain the number of the lowest variable whose instantiation has changed since the variable x_3 was last instantiated with a new value, in this case 2. Yet, the value of *mbl*[3] is 1 (see Figure 4.13), and so the same check is performed again.

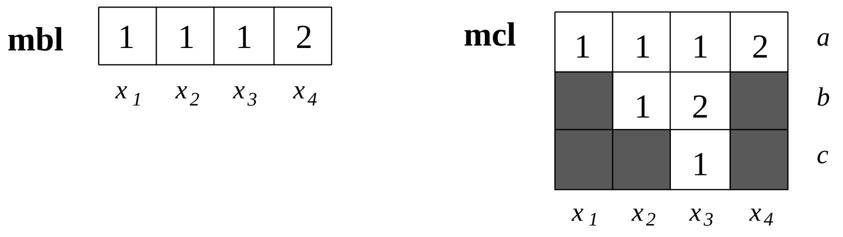


Figure 4.13: Arrays *mbl* and *mcl* of BMJ before node 7 is visited.

The value is of *mbl*[3] is not entirely incorrect however, as can be seen at node 10, which corresponds to tuple $((x_1, a), (x_2, b), (x_3, c))$. BM and BMJ behave differently at this node. BM "knows" that instantiation (x_3, c) is consistent with instantiation (x_1, a), because it performed the consistency check at node 6. BMJ, however, skipped node 6, and so has to perform this consistency check now.

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

4.1.2 BMJ2 — A Modified BMJ

The modified BMJ, which we call BMJ2, 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 1 in the beginning, and then 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. The BMJ2 code is presented below along with the code of BMJ.

```
int consistent(z)                   =  int consistent(z)
int z;                              =  int z;
{                                   =  {
  int i, olmbl;                     =    int i, oldmbl;
                                    =
  oldmbl = mbl[z];                  #    oldmbl = mbl[z][v[z]];
                                    #    mbl[z][v[z]] = z;
  if (mcl[z][v[z]] < oldmbl)        =    if (mcl[z][v[z]] < oldmbl)
    return(0);                      =      return(0);
  for (i = oldmbl; i < z; i++) {    =    for (i = oldmbl; i < z; i++) {
    mcl[z][v[z]] = i;               =      mcl[z][v[z]] = i;
    if (check(z,i) == 0) {          =      if (check(z,i) == 0) {
      max_check[z] =                =        max_check[z] =
          max(max_check[z],i);      =            max(max_check[z],i);
      return(0); } }                =        return(0); } }
  max_check[z] = z - 1;             =    max_check[z] = z - 1;
  return(1);                        =    return(1);
}                                   =  }
                                    =
int BMJ(z)                          #  int BMJ2(z)
int z;                              =  int z;
{                                   =  {
  int h, i, jump;                   #    int h, i, j, jump;
                                    =
  if (z > N) {                      =    if (z > N) {
    solution();                     =      solution();
    return(N); }                    =      return(N); }
  max_check[z] = 0;                 =    max_check[z] = 0;);
  for (i = 0; i < K; i++) {         =    for (i = 0; i < K; i++) {
    v[z] = i;                       =      v[z] = i;
    if (consistent(z)) {            =      if (consistent(z)) {
      jump = BMJ(z + 1);            =        jump = BM_CBJ(z + 1);
      if (jump != z)                =        if (jump != z)
        return(jump); } }           =          return(jump); } }
  h = max_check[z];                 =    h = max_check[z];
  mbl[z] = h;                       #
  for (i = h+1; i <= N; i++)        =    for (i = h+1; i <= N; i++)
                                    #      for (j = 0; j < K; j++)
    mbl[i] = min(mbl[i],h);         #        mbl[i][j] = min(mbl[i][j],h);
  return(h);                        =    return(h);
}                                   =  }
```

Let us now analyze the behaviour of the modified algorithm on our example. The *mcl* array (Figure 4.14) looks exactly as in the case of BMJ, but *mbl* is now 2-dimensional. After the left subtree is visited, $mbl[3,a]$ and $mbl[3,b]$ are set to 2, but $mbl[3,c]$, which corresponds to the skipped node 6, remains unchanged at 1. Savings are then made at nodes 8, 9 and 11.

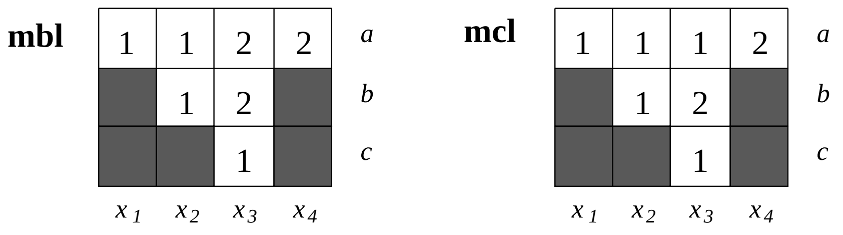


Figure 4.14: Arrays *mbl* and *mcl* of BMJ2 before node 7 is visited.

Table 4.2 contains a node-by-node comparison of all algorithms discussed in this section. The only node at which BMJ2 performs more consistency checks than any other algorithm is node 10. However, note that the extra check ((1a,3c)) is performed by BM earlier, at node 6.

	node	BT	BJ	BM	BMJ	BMJ2
1	a	0	0	0	0	0
2	aa	1	1	1	1	1
3	aaa	1	1	1	1	1
4	aab	2	2	2	2	2
5	aaba	2	2	2	2	2
6	aac	2	-	2	-	-
7	ab	1	1	1	1	1
8	aba	1	1	0	1	0
9	abb	2	2	1	2	1
10	abc	2	2	1	2	2
11	abca	3	3	2	2	2
$\sum$		17	15	13	14	12

Table 4.2: Number of consistency checks performed at each node by various backtracking algorithms.

An analogous modification of BM-CBJ produces BM-CBJ2: *mbl* should be made a 2-dimensional array, and maintained in the same way as in BMJ2.

4.2 GBJ

In **Graph-Based Backjumping (GBJ)** the backtrack tree is determined by the topology of the constraint network. In contrast with BJ and CBJ, the actual constraints have no influence on the backtracking behaviour of this algorithm. GBJ always backtracks to the most recent variable connected to the current variable in the constraint network. The topological information is computed once at the start of algorithm and stored in the PARENTS sets, one set for each variable. This approach results in small overhead costs, but considerable savings are achieved only if the constraint network is sparse. For a fully connected network, such as in the n-queens problem, GBJ generates the same backtrack tree as BT.

The behaviour of GBJ is similar to the behaviour of CBJ. The main difference is that the static PARENTS sets are used instead of the dynamic conflict sets. Since the existence of a nontrivial constraint between two variables does not imply a conflict between their instantiations, we may expect GBJ to perform shorter backjumps and consequently visit more nodes than CBJ.

In order to make GBJ find all solutions, the same modification as in the case of CBJ must be made. A one-dimensional array should be employed to differentiate between backtracking from an inconsistency, and backtracking after finding a solution (see Chapter 3.3).

The backjumping behaviour of GBJ may be described by a lemma analogous to Lemma 2. The only difference in the lemma and its proof would be to use PARENTS sets instead of conflict sets.

In order to partially characterize the set of nodes visited by GBJ, the backjumping lemma may be used to formulate two theorems, analogous to Theorems 3 and 6. The first one, analogous to Theorem 3, states that GBJ visits a node if its parent is consistent with all sets of variables. The second one, analogous to Theorem 6, states that GBJ visits a node only if its parent is consistent.

Note that proving such two theorems about any static-order backtracking algorithm amounts to proving its correctness. The first theorem states that all nodes that lie on the paths between root and solutions are visited, which guarantees that no solution is omitted. The second theorem states that only consistent nodes are expanded, which that guarantees the solutions claimed by the algorithm are consistent. The two theorems constitute the minimal characterization of the set of nodes visited by a backtracking algorithm. The characterization is strong enough, however, to prove the algorithm's correctness (see the proof of Theorem 7).

In the hierarchies, GBJ may probably be placed between BT and CBJ. It is clear that BT visits all nodes that GBJ visits. Also, the similarity between PARENTS sets and conflict sets and experimental results suggest that GBJ visits all nodes that CBJ visits. However, since we have not been able to prove it, this proposition remains a conjecture. BT, GBJ, and CBJ perform the same number of consistency checks at each visited node; therefore, their ordering with respect to the number of consistency checks is the same as their ordering with respect to nodes.

4.3 FC Hybrids

Two FC hybrids have not yet been discussed. In this section, we briefly state our results informally in points.

1. A similar modification as in the case of CBJ must be made to FC-CBJ in order to make it find all solutions.

2. FC-CBJ visits a node only if it is consistent and its parent is consistent with all variables (this is the same necessary condition as for FC).
3. FC-CBJ visits a node if it is consistent and its parent is consistent with all sets of variables (this is the same sufficient condition as for CBJ).
4. FC-CBJ is correct (implied by Point 2 and 3).
5. FC visits all nodes that FC-CBJ visits (implied by Point 2).
6. At any visited node FC-CBJ performs the same number of consistency checks as FC.
7. FC-CBJ performs no more consistency checks than FC (follows from Points 5 and 6).

All the above observations except the first one apply to FC-BJ as well. Also, we conjecture that FC-CBJ always skips more nodes that FC-BJ, and consequently performs less consistency checks.

4.4 Hierarchies

We can now expand the hierarchy given in the Chapter 3 (Figure 3.9) to include the backtracking algorithms studied in this chapter.

We have noted that imposing a marking scheme on a backtracking algorithm does not change the set of nodes which are visited. A marking scheme causes an algorithm to avoid some of the redundant consistency checks, but it has no influence on the algorithm's search tree. Therefore, the sets of nodes expanded by the following algorithms are identical:

- BT and BM
- BJ and BMJ (BMJ2)
- CBJ and BM-CBJ (BM-CBJ2)

The final hierarchy, which includes the observations made in the previous two sections, is presented in Figure 4.15. The hard links represent formally proven or obvious relationships. The soft links represent conjectures which are suggested by analyses and experimental results.

The second hierarchy seems to be even more important because the number of consistency checks is a measure that better reflects the actual run times.

Let us define a relation $\leq_{cc}$ (partial order) between backtracking algorithms.

Definition 3 *$\mathcal{A} \leq_{cc} \mathcal{B}$ if and only if a backtracking algorithm $\mathcal{A}$ performs no more consistency checks than a backtracking algorithm $\mathcal{B}$ when finding all solutions of any constraint satisfaction network.*

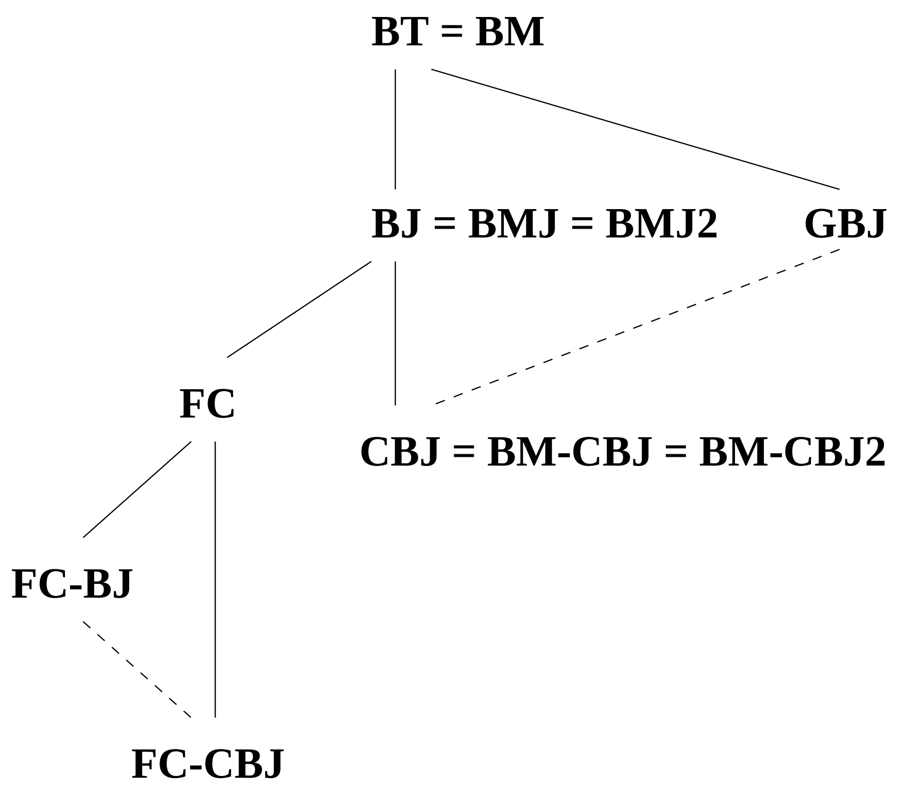


Figure 4.15: The hierarchy with respect to the number of visited nodes.

In addition to the relationships discussed in Sections 4.2 and 4.3, the relation contains the following pairs:

BJ $\leq_{cc}$ **BT** From Corollary 1 we know that BT visits all nodes that BJ visits. At any given node both algorithms perform the same number of consistency checks. Therefore, BJ performs no more consistency checks than BT on the whole network.

CBJ $\leq_{cc}$ **BJ** From Theorem 9 we know that BJ visits all nodes that CBJ visits. At any given node both algorithms perform the same number of consistency checks. Therefore, CBJ performs no more consistency checks than BJ on the whole network.

BM $\leq_{cc}$ **BT** BT and BM generate identical backtrack trees. However, thanks to the marking scheme, at any given node BM performs no more consistency checks than BT. Therefore, BM performs no more consistency checks than BT on the whole network.

BMJ $\leq_{cc}$ **BJ** BJ and BMJ generate identical backtrack trees. However, thanks to the marking scheme, at any given node BMJ performs no more consistency checks than BJ. Therefore, BMJ performs no more consistency checks than BJ on the whole network.

BM-CBJ $\leq_{cc}$ **CBJ** Similar argument as the previous one.

Experiments and analyses suggest also the following conjectures:

- **BMJ2** $\leq_{cc}$ **BMJ**
- **BM-CBJ2** $\leq_{cc}$ **BM-CBJ**
- **BM-CBJ2** $\leq_{cc}$ **BMJ2** $\leq_{cc}$ **BM**

Figure 4.16 presents the hierarchy of algorithms with respect to the number of consistency checks. Besides the relationships that are shown explicitly, it is important to note the ones which are *not* 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 more consistency checks than FC on the confused 12-queens problem, but less on the regular 12-queens problem (Table 5.3). Examples of constraint networks can be found that disprove all relationships which are not included in the hierarchies. Thus, however counterintuitive it may seem, FC-CBJ may visit more nodes than GBJ, and perform more consistency checks than BT[1].

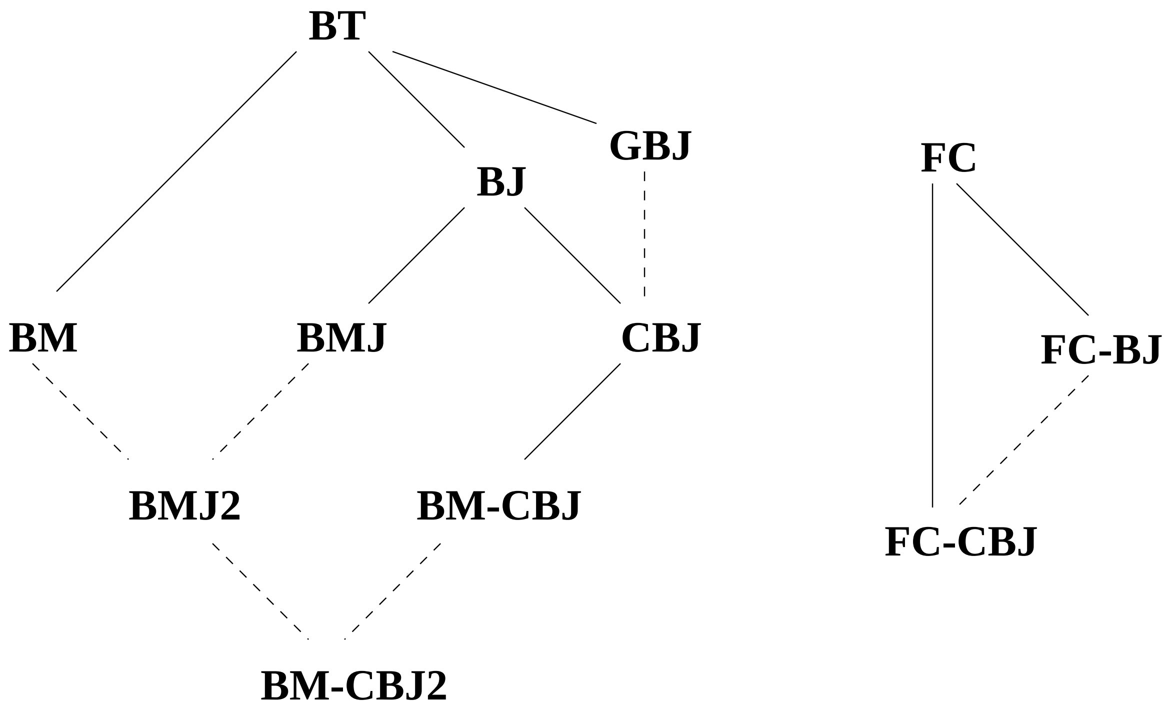


Figure 4.16: The hierarchy with respect to the number of consistency checks.

[1]Prosser [17] gives an example of a problem on which BT outperforms any algorithm based on forward checking.

Chapter 5

Experimental Results

In spite of the strongly theoretical approach adopted in this work, we include a handful of experimental results. Three well-known benchmark problems and one randomly generated problem were chosen for the comparison of ten backtracking algorithms (the CLGP versions).

As benchmark problems, we used the regular 12-queens problem, the confused 40-queens problem, and the zebra problem. The queens problems have already been described in previous chapters. We adopted the same, normal ordering as given in Examples 1, 2, and 3. The zebra problem has 25 variables with domains of size 5. We used the problem formulation and the ordering defined by Dechter in [4]. This variant of the zebra problem has one solution. The results on the benchmark problems may be reproduced in order to verify the equivalence of other implementations of the same backtracking algorithms.

The random problem was generated using a function from the CSP code library [12]. The generator has two parameters: the probability of a nontrivial constraint between two variables, which was set at $p = 0.12$, and the probability of an allowable pair in a constraint, which was set at $q = 0.22$. The problem has 15 variables with domains of size 10, and has no solution. An effort was made to select the values of the parameters p and q so that they are close to the boundary that separates the overconstrained (no solution) networks from the underconstrained (many solutions) networks (as described in [3]). The resulting problem is computationally hard.

The results in Tables 5.3 and 5.4 show that the relative performance of the algorithms varies dramatically on different problems. On the regular queens, BM and FC perform well, but the additional backjumping ability does not improve their performance significantly. This is because the density of constraints is high (every two variables are connected), and so long backjumps are rare. On the confused queens, there is very little difference between the worst and the best results in terms of consistency checks. The smallest backtrack trees are generated by FC and its hybrids. The problem is rather easy; therefore, it does not require sophisticated techniques. The performance of the algorithms on the zebra problem depends heavily on the variable ordering (for an excellent statistical analysis see [17]). On this instance of the problem, BM and its hybrids are the best. In the case of the hard random problem,

	REG Q	CONF Q	ZEBRA	RANDOM
BT	45,396,914	181,300	32,635	792,670,770
BJ	38,511,567	151,129	30,930	178,238,158
CBJ	36,890,689	151,129	15,331	101,368
GBJ	45,396,914	181,300	31,331	2,385,869
BM	5,224,512	115,640	7,300	44,971,390
BMJ	5,309,340	123,698	7,204	13,609,324
BMJ2	5,003,276	114,557	7,002	13,326,594
BM-CBJ	5,306,272	123,698	5,454	49,415
BM-CBJ2	4,938,324	114,557	5,139	31,444
FC	5,958,644	98,696	11,060	611,118
FC-BJ	5,923,788	98,696	8,186	250,013
FC-CBJ	5,915,759	98,696	8,186	40,824

Table 5.3: Number of consistency checks performed by various backtracking algorithms on certain constraint satisfaction problems.

	REG Q	CONF Q	ZEBRA	RANDOM
BT	10,103,868	127,880	3,488	202,166,510
BJ	8,545,890	98,902	2,990	38,566,291
CBJ	8,176,526	98,902	1,673	23,450
GBJ	10,103,868	127,880	3,296	586,545
BM	10,103,868	127,880	3,488	202,166,510
BMJ	8,545,890	98,902	2,990	38,566,291
BMJ2	8,545,890	98,902	2,990	38,566,291
BM-CBJ	8,176,526	98,902	1,673	23,450
BM-CBJ2	8,176,526	98,902	1,673	23,450
FC	641,974	1,756	511	32,742
FC-BJ	629,854	1,756	307	10,492
FC-CBJ	627,997	1,756	307	911

Table 5.4: Number of nodes visited by various backtracking algorithms on certain constraint satisfaction problems.

the difference between the algorithms that use multiple backjumps and the other algorithms is stark. The best results are produced by the hybrids that combine CBJ with BM or FC.

In summary, the empirical results confirm our theoretical findings. The relative performance of the algorithms varies significantly; on some problems the hybrid algorithms are much better than the basic algorithms, whereas on other problems the differences are negligible. However, the rankings of the algorithms always agree with the partial orders we give in Chapter 4. As for the modified hybrids, on all four problems BM-CBJ2 is the best of the nine backward checking algorithms, and BMJ2 is better than either BM or BJ.

Chapter 6

Conclusions

In this chapter we provide some suggestions for future work, and a summary.

6.1 Future Work

1. The characterizing conditions for all algorithms except BT and FC do not cover all nodes in their backtrack trees. Ideally, we would like the sufficient and the necessary conditions to be the same. Since backtracking algorithms are deterministic, it seems that it should be possible to describe precisely their backtrack trees.

2. There exist many other backtracking algorithms which have not been treated in this thesis such as algorithms with variable ordering, and algorithms that combine consistency enforcing techniques with backtracking. Our approach could be applied to all those algorithms.

3. 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.

4. The conjectures concerning GBJ and FC hybrids are yet to be proven formally.

6.2 Summary

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 which 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 a hierarchy of algorithms with respect to the number of visited nodes. Next, we constructed a hierarchy of algorithms with respect to the number of consistency checks, which is a better performance measure than number of nodes. 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. The empirical tests showed one of the modified algorithms to be better (in terms of consistency checks) than all six backward checking algorithms described by Prosser in [17].

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

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

[3] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 331–337, 1991.

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

[5] R. Dechter. Constraint networks. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence, Second Edition.* John Wiley & Sons, 1992.

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

[7] 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, 1978.

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

[9] S. W. Golomb and L. D. Baumert. Backtrack programming. *Journal of the ACM*, 12:516–524, 1965.

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

[11] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[12] D. Manchak and P. van Beek. A ‘C’ library of constraint satisfaction techniques, 1994. Available by anonymous ftp from: ftp.cs.ualberta.ca:pub/ai/csp.

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

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

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

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

[17] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. Technical Report AISL–46–91, University of Strathclyde, September 1991.

[18] P. Prosser. Domain filtering can degrade intelligent backtrackng search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 262–267, 1993.

[19] P. Prosser. Forward checking with backmarking. Technical Report AISL–48–93, University of Strathclyde, June 1993.

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