Computational Intelligence, Volume 9, Number 3, 1993

HYBRID ALGORITHMS FOR THE CONSTRAINT SATISFACTION PROBLEM

PATRICK PROSSER
Department of Computer Science
University of Strathclyde, Livingstone Tower
Glasgow G1 1XH, Scotland
e-mail: pat@cs.strath.ac.uk

It might be said that there are five basic tree search algorithms for the constraint satisfaction problem (csp), namely, naive backtracking (BT), backjumping (BJ), conflict-directed backjumping (CBJ), backmarking (BM), and forward checking (FC). In broad terms, BT, BJ, and CBJ describe different styles of backward move (backtracking), whereas BT, BM, and FC describe different styles of forward move (labeling of variables). This paper presents an approach that allows base algorithms to be combined, giving us new hybrids. The base algorithms are described explicitly, in terms of a forward move and a backward move. It is then shown that the forward move of one algorithm may be combined with the backward move of another, giving a new hybrid. In total, four hybrids are presented: backmarking with backjumping (BMJ), backmarking with conflict-directed backjumping (BM-CBJ), forward checking with backjumping (FC-BJ), and forward checking with conflict-directed backjumping (FC-CBJ). The performances of the nine algorithms (BT, BJ, CBJ, BM, BMJ, BM-CBJ, FC, FC-BJ, FC-CBJ) are compared empirically, using 450 instances of the ZEBRA problem, and it is shown that FC-CBJ is by far the best of the algorithms examined.

Key words: constraint satisfaction problem, tree search algorithms, backtracking, backjumping, backmarking, forward checking.

1. INTRODUCTION

The work reported in this paper was motivated by the following questions posed by Nadel (1989):

> Something to think about would be a synthesis of BM and BJ, into an algorithm called, say, BMJ (BackMarkJump). . . . Is it possible to combine both approaches while retaining all, or most, of the power of each?

and further:

> Combining *j*-consistency with Backjump or Backmark should be possible, as suggested by Gaschnig. And Backmark and Backjump may themselves perhaps be combined. . . . Such algorithms deserve attention.

This paper presents four "hybrid" tree search algorithms (algorithms created by combining the forward move of one algorithm with the backward move of another) for the constraint satisfaction problem (csp), one of these being BMJ. In addition, an algorithm which combines 2-consistency with backjumping is also presented. The technique of combining algorithms is presented, along with an empirical analysis of nine tree search algorithms.

There appear to be five basic tree search algorithms for the constraint satisfaction problem, namely naive backtracking (BT) (Golomb and Baumert 1965), backjumping (BJ) (Gaschnig 1979), conflict-directed backjumping (CBJ, a new algorithm described later on), backmarking (BM) (Gaschnig 1977, 1979), and forward checking (FC) (Haralick and Elliott 1980). In broad terms these algorithms might be classified as follows: BT, BM, and FC

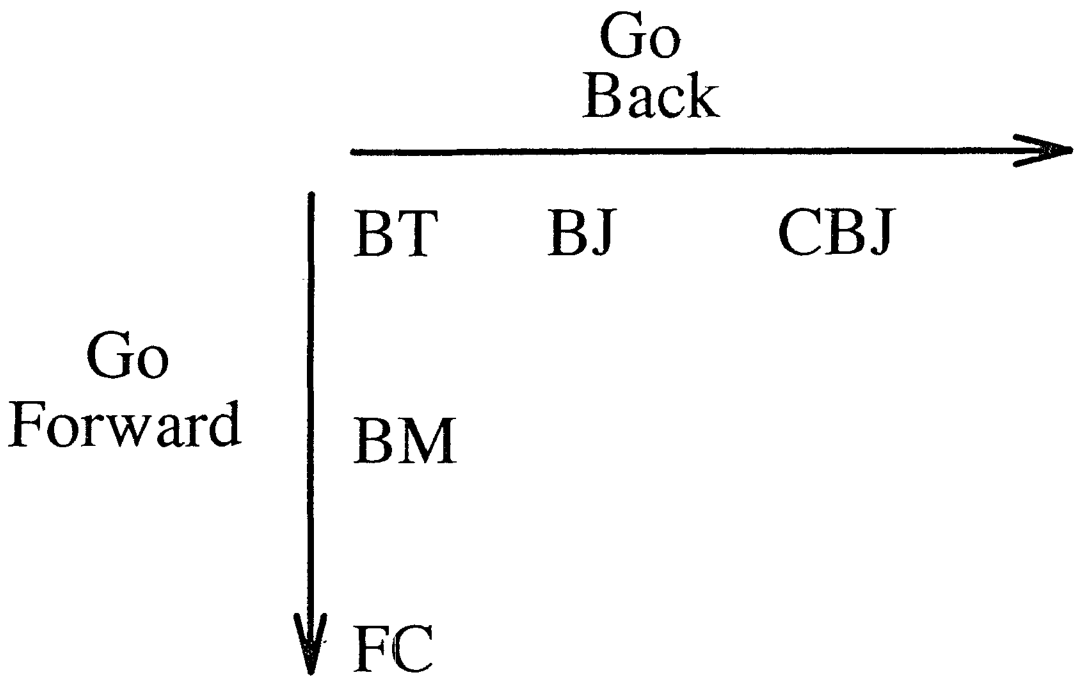

FIGURE 1. The five base algorithms.

chronologically backtrack, whereas BJ and CBJ are informed backtrackers. BT, BM, BJ, and CBJ check backwards, from the current variable against the past variable, whereas FC checks forward, from the current variable against the future variables.

The algorithms may be viewed from a different perspective. When we move from BT → BJ → CBJ we move progressively toward more informed styles of backtracking. However, BT, BJ, and CBJ all use the same style of forward move (labeling of variables). When we move from BT → BM → FC we traverse across different styles of forward move, but again each of these algorithms use the same style of backward move (chronological backtracking). Therefore BT, BM and FC essentially describe a style of forward move, and BT, BJ, and CBJ describe a style of backward move.[1]

In Fig. 1 we have these five base algorithms. To top row represents moves, and the first column represents forward moves. It appears that four algorithms are missing. We should expect that we can take the forward move of one algorithm (for example FC) and combine it with the backward move of another (for example BJ) to give a new "hybrid" algorithm (for example FC-BJ, an algorithm that checks forward and jumps back). Therefore, we should expect the nine algorithms of Fig. 2.

In Fig. 2, algorithms in a given row exploit the same style of forward move, and algorithms in a given column exploit the same style of backward move. When we move across the row (left to right) we move toward more informed styles of backtracking, and when we move down a column we move across different styles of foward move.

Historically, tree search algorithms for the csp have been described in a recursive style, such that a recursive call corresponds to a foward move, and a return from a call

[1]We should consider BT as describing the most primitive forward move (checking against past variables) and the most primitive backward move (chronological backtracking).

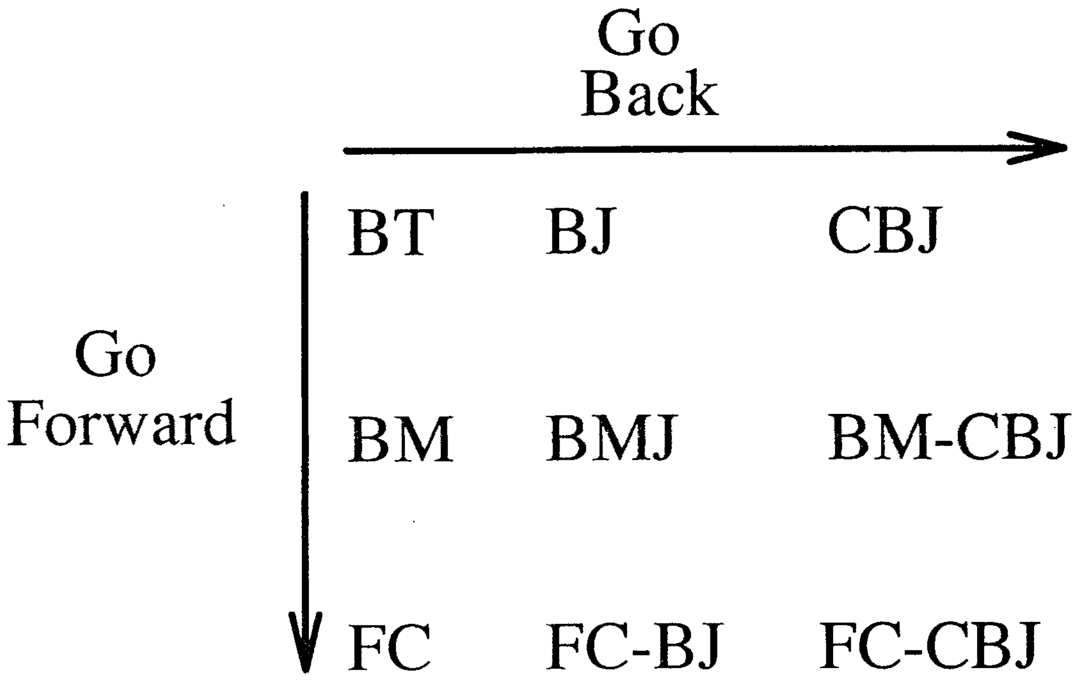

FIGURE 2. The five base and four hybrid algorithms.

corresponds to a backward move. Therefore, the forward and backward moves are described implicitly, and search knowledge may be hidden within the procedure stack. This paper adopts a different approach. A tree search algorithm *X* is described by two functions, *x-label* and *x-unlabel*, and a calling procedure. Function *x-label* corresponds to the foward move of *X*, and *x-unlabel* corresponds to the backward move of *X*. The functions are then called iteratively by a procedure (in this case *bcssp*, described in the following section). Therefore the forward and backward moves are made explicit, as is the search knowledge.[2] The act of combining algorithms is therefore simplified. To synthesize the hybrid *X-Y* we take the forward move of *X*, *x-label*, and modify it such that it maintains the information required by the backward move of *Y*, giving us the function *x-y-label*. In addition we take the backward move of *Y*, *y-unlabel*, and modify it such that it maintains the information required by the forward move of *X*, giving us the function *x-y-unlabel*. The two functions, *x-y-label* and *x-y-unlabel*, then describe the hybrid *X-Y*.

The remainder of this paper is organized as follows. The next section introduces the constraint satisfaction problem and the terminology applied to that problem. The coding conventions are introduced, along with the global variables that will be used by the following algorithms. Section 3 describes nine tree search algorithms for the csp. Section 4 describes the experiments that were performed, and Section 5 analyses these results. Section 6 concludes this paper, looking backward over what has been presented, and forward toward what might still be done.

[2]This approach is not new, as it was used by Dechter when describing chronological backtracking (Dechter and Pearl 1988) and graph-based backjumping (Dechter 1990). However, it was not exploited as a technique for combining algorithms, such as BM with BJ.

2. DEFINITIONS AND PROGRAMMING CONVENTIONS

Definition 1. The *binary constraint satisfaction problem* (bcsp) involves a set of variables $\{V_1, V_2, \ldots, V_n\}$. Each variable V_i has a finite (and discrete) domain of values $D_i = \{v_{i1}, v_{i2}, \ldots, v_{iM_i}\}$ and may be assigned any one of the values $v_{ij} \in D_i$. In addition, we have a set of binary constraints $\{C_{1,1}, C_{1,2}, \ldots C_{1,n}, \ldots, C_{2,1}, C_{2,2} \ldots, C_{2,n}, \ldots, C_{n,1}, C_{n,2} \ldots, C_{n,n}\}$, where the constraint $C_{i,j}$ is a relation between V_i and V_j, and if $C_{i,j}$ is null then there is no constraint acting from V_i to V_j. A binary constraint satisfaction problem can be associated with a constraint graph G (Mackworth 1977). $V(G)$, the set of vertices in G, corresponds to the set of variables, and $A(G)$, the set of directed arcs in G, corresponds to the set of binary constraints. The problem is then to find an assignment of values to variables, from their respective domains, which satisfy the constraints. There are a number of variants of this problem (Nudel 1983). The one addressed in this paper is the binary constraint satisfaction "search" problem (bcssp). That is, we attempt to find the first solution. For the sake of brevity, the bcsp and its variant the bcssp will from now on be referred to as the csp.[3]

Definition 2. The *order of instantiation* is the order in which variables are assigned values. The order of instantiation may be static or dynamic. In a static instantiation order the search process always instantiates some variable V_i before some other variable V_j. In a dynamic instantiation order the search process decides which variable to instantiate next based on the state of the search process. In this study we assume a static order of instantiation.

Definition 3. The *current* variable is the variable chosen for instantiation. Generally V_i will be considered as the current variable.

Definition 4. The *past* variables are the variables that have already been instantiated. If variable V_h was instantiated before variable V_i it may be said that V_h is in V_i's past. This may be represented via the ordering relation $h < i$. Therefore, we assume that the search tree grows downward and that V_1 is the root. Variables near the root of the search tree are then at a "shallow" depth and have low-valued subscripts, and variables far from the root are "deep" and have high-valued subscripts.

Defintion 5. The *future* variables are the variables that have not yet been instantiated. If variable V_i was instantiated before variable V_j it may be said that V_j is in V_i's future. This may be represented by the ordering relation $j > i$.

The algorithms that follow are described in a pseudocode modeled on Pascal and Common Lisp and that is an enhancement of that given in Nadel (1989). A fuller description of this language is given in Nadel (1989) and Appendix A of this paper. The following assumptions are made.

- The language supports list processing. It is assumed that the list processing functions *list, push, pop, pushnew, remove, set-difference, union*, and *max-list* are primitives of the language.

[3]For a broader introduction to the constraint satisfaction problem one might work through Meseguer's overview (Meseguer 1989), Kumar's survey (Kumar 1992), and the encyclopedia entries of Dechter (Dechter 1992) and Mackworth (Mackworth 1992).

- Variables local to a procedure are implicitly declared. The first occurrence of a variable within a procedure corresponds to an implicit declaration of that variable.

The following variables are assumed to have been globally declared and thus accessible to all procedures.

v: *v* is an array of values, such that *v*[*i*] is the value assigned to the variable V_i. From now on we will use *v*[*i*] in place of V_i when referring to the *i*th variable.

n: *n* is the number of variables actually in the problem. We assume that the first variable is *v*[1] and the last variable is *v*[*n*]. We also assume the existence of the pseudovariable *v*[0]. This is used as a convenience, i.e., an attempt to backtrack to *v*[0] will result in termination of the search process.

domain: *domain* is an array of sequences, such that *domain*[*i*] (for $0 \leq i \leq n$) is the domain of the variable *v*[*i*]. *domain*[*i*] is synonymous with D_i, where *domain*[*i*] is a finite sequence of discrete values. Note the pseudovariable *v*[0] has *domain*[0] = *nil*.

current-domain: *current-domain* is an array of sequences. *current-domain*[*i*] (for $0 \leq i \leq n$) is the sequence of values in *domain*[*i*] that have not yet been shown to be inconsistent with respect to the ongoing search process. *current-domain*[*i*] is initialized to be equal to *domain*[*i*] (consequently *current-domain*[0] = *nil*). When the search process attempts to instantiate *v*[*i*] with a value, it selects that value from *current-domain*[*i*]. If that value is found to be incompatible with the current search state, it is then removed from *current-domain*[*i*]. If *current-domain*[*i*] is empty (nil), then the search process has examined all possible instantiations for *v*[*i*] without success, and backtracking takes place. When backtracking takes place (generally) *current-domain*[*i*] is reinstated (i.e., it becomes *domain*[*i*] again).

C: *C* is an $n \times n$ array, where *C*[*i,j*] is the name of a binary predicate (such as $<$, $=$, $>$, etc.) that holds between *v*[*i*] and *v*[*j*]. If *C*[*i,j*] = *nil* then there is no constraint acting between *v*[*i*] and *v*[*j*], and all values in *domain*[*i*] are compatible with all values in *domain*[*j*]. Therefore, we have an *intensional* representation of a constraint, rather than an *extensional* representation of a constraint (as a set of compatible pairs). We might think of *C* as being a richer representation of the adjacency matrix of a directed graph. Rather than being a count of the number of directed arcs from vertex *i* to vertex *j*, *C*[*i,j*] is the name of a binary predicate (or nil).

check(*i,h*): The function *check*(*i,j*) delivers a result of *true* if there is no constraint between *v*[*i*] and *v*[*j*] (that is *C*[*i,j*] = *nil*); otherwise it delivers the result of applying the binary predicate *C*[*i,j*] between the instantiations of *v*[*i*] and *v*[*j*] (and is counted as a consistency check).

The procedure below, *bcssp*, shows the environment within which the tree search functions will be called.

```
PROCEDURE bcssp (n,status)
BEGIN
  consistent ← true;
  status ← "unknown";
  i ← 1;
  WHILE status = "unknown"
  DO BEGIN
    IF consistent
    THEN i ← label(i,consistent)
    ELSE i ← unlabel(i,consistent);
```

```
11      IF i > n
12      THEN status ← "solution"
13      ELSE IF i = 0
14        THEN status ← "impossible"
15      END
16  END;
```

Procedure *bcssp* addresses the binary constraint satisfaction search problem:

Given a set of variables (where each variable has a discrete domain) and a set of binary relations that act between pairs of variables, find the first consistent instantiation of these variables which satisfies all the relations.

The function *label* attempts to find a consistent instantiation for $v[i]$. The function takes as arguments the boolean (reference) variable *consistent* and the current variable i. *label* is called when *consistent* = *true* and $1 \le i \le n$. The function delivers as a result the new current variable. When *label* terminates with *consistent* = *true*, the variable $v[i]$ will have been instantiated with a value that is consistent with the past variables, and $i+1$ is delivered as a result (thus maintaining the static order of instantiation). When *label* terminates with *consistent* = *false* then no consistent instantiation could be found for $v[i]$, *current-domain*$[i]$ = *nil*, and i is delivered as a result. Therefore, *label* can terminate in the following states: (a) *consistent* = *true* and $1 \le i \le n$, (b) *consistent* = *true* and $i = n+1$, or (c) *consistent* = *false* and $1 \le i \le n$. When terminating in state (a) procedure *bcssp* will again call *label* with the new current variable. Terminating in state (b) will cause procedure *bcssp* to terminate with *status* set to *solution*, and terminating in state (c) will cause *bcssp* to call *unlabel*.

Function *unlabel* performs backtracking from $v[i]$ to $v[h]$. The function is called when *consistent* = *false*, $1 \le i \le n$, and *current-domain*$[i]$ = *nil* (all values have been tried for v[i] without success). The function selects a past variable $v[h]$ as the backtracking point and resets the variable $v[j]$, for all j, where $h < j \le i$. The value in $v[h]$ is then removed from *current-domain*$[h]$, *consistent* is set to *true* if there are values remaining in *current-domain*$[h]$, and h is delivered as a result. Therefore *unlabel* can terminate in the following states: (a) *consistent* = *true* and $1 \le h \le i$, (b) *consistent* = *false* and $1 \le h < i$, or (c) *consistent* = *false* and $h = 0$. When terminating in state (a) procedure *bcssp* will then call *label*. Terminating in state (b) will cause procedure *bcssp* to call *unlabel* again, and terminating in state (c) will cause *bcssp* to terminate with *status* = *impossible*.

In the functions that follow, a forward move by algorithm X will be named *x-label*, and a backward move will be named *x-unlabel*. For example BT (chronological backtracking) is defined by functions *bt-label* and *bt-unlabel*. These functions are then substituted into lines 9 and 10 respectively, of procedure *bcssp*. In the experiments that follow, the number of calls made to *x-label* is taken to be the number of nodes visited within the search tree.

3. TREE SEARCH ALGORITHMS

This section describes nine tree search algorithms for the constraint satisfaction search problem, and these are presented in the following order: BT, BJ, CBJ, BM, BMJ, BM-CBJ, FC, FC-BJ, FC-CBJ. Generally, an algorithm is presented as a modification to an existing algorithm, and line numbering is adopted so that we can see just what changes are required to take us from one algorithm to another. In essence, all the algorithms are arrived at by performing minor modifications to BT, and this algorithm might be considered as a reference point.

3.1. Chronological Backtracking

In the backtracking algorithm (Bitner and Reingold 1975; Golomb and Baumert 1965; Walker 1960) variables are incrementally instantiated with values from their respective domains. When the current variable *v*[*i*] is assigned a value, consistency checking is performed backward against the past variables. If a consistency check fails then another value is selected from the domain of *v*[*i*] and consistency checking is performed again. If no value can be found in the domain of *v*[*i*] that is consistent with the past variables, then the variable *v*[*h*], which was instantiated immediately before *v*[*i*], is uninstantiated and a new value is sought for *v*[*h*]. The backtracking algorithm BT is described below by the functions *bt-label* and *bt-unlabel.*

```
 1  FUNCTION bt-label(i,consistent): INTEGER
 2  BEGIN
 3    consistent ← false;
 4    FOR v[i] ← EACH ELEMENT OF current-domain[i] WHILE not consistent
 5    DO BEGIN
 6      consistent ← true;
 7      FOR h ← 1 TO i−1 WHILE consistent
 8      DO consistent ← check(i,h);
 9      IF not consistent
10      THEN current-domain[i] ← remove(v[i],current-domain[i])
11      END;
12    IF consistent THEN return(i+1) ELSE return(i)
13  END;
```

bt-label attempts to find the first instantiation in *current-domain*[*i*] that is consistent with all of the past variables (the FOR loop, lines 4–11). When *bt-label* encounters some value in *current-domain*[*i*] that is inconsistent with the past variables, that value is removed from *current-domain*[*i*] (lines 9 and 10). The outer FOR loop terminates either (a) by making a consistent instantiation of *v*[*i*] or (b) by exhausting all values in *current-domain*[*i*]. When terminating in state (a) *consistent* will be true, *current-domain*[*i*] $\subseteq$ *domain*[*i*], and $i+1$ is delivered as the new current variable. When terminating in state (b), *current-domain*[*i*] will be *nil*, *consistent* will be *false*, and *i* is delivered as the current variable.

```
1  FUNCTION bt-unlabel(i,consistent): INTEGER
2  BEGIN
3    h ← i−1;
4    current-domain[i] ← domain[i];
5    current-domain[h] ← remove(v[h],current-domain[h]);
6    consistent ← current-domain[h] ≠ nil;
7    return(h)
8  END;
```

bt-unlabel chronologically backtracks from *v*[*i*] to *v*[*h*], where $h \leftarrow i-1$ (line 3). As will be seen, line 3 is common to all of the chronological backtracking functions (*bt-unlabel, bm-unlabel,* and *fc-unlabel*). *current-domain*[*i*] is reset to *domain*[*i*] (line 4) and the value *v*[*h*] is removed from *current-domain*[*h*] (line 5). When *v*[*h*] becomes the current variable (line 7) all future variables, *v*[*j*], will have *current-domain*[*j*] = *domain*[*j*], and all past variables (and the current variable) will have *current-domain*[*g*] $\subseteq$ *domain*[*g*], where $1 \leq g \leq h < j \leq n$. This property holds for all of the backward-checking algorithms (BT, BJ, CBJ, BM, BMJ, and BM-CBJ). The reference variable *consistent* is set to *true* if there are values remaining in *current-domain*[*h*] (i.e., BT can now attempt a new instantiation for *v*[*h*]; otherwise *consistent* is set to *false* (i.e., BT will then backtrack from *v*[*h*] to *v*[*h*−1]). The function returns *h* as the new current variable. The actions of lines 5, 6, and 7 are common to all of the backtracking functions presented here. The action of line 5 in *bt-unlabel* might be interpreted as follows.

> Function *bt-label* was unable to find an instantiation for *v*[*i*] which was consistent. It is assumed that the instantiation of *v*[*h*] is the cause of this inconsistency. Therefore, by finding a new instantiation for *v*[*h*], consistent instantiations might be found for the future variables.

This is a naive assumption. It may be the case that *v*[*h*] plays no role whatsoever in the conflict involving *v*[*i*]. When this happens the entire search subtree rooted on *v*[*i*] will be reexplored, and the functions *bt-label* and *bt-unlabel* will slavishly repeat the same set of actions with the same set of outcomes. This pathological behavior has been referred to as thrashing (Mackworth 1977).

3.2. Backjumping (BJ)

The backjumping (BJ) procedure of Gaschnig (1979) attempts to minimize the number of nodes visited within the search tree and consequently reduce the number of consistency checks performed by the search process. BJ does this by jumping back directly to the cause of a conflict. When the current variable *v*[*i*] is to be instantiated with a value, a record is kept in the array element *max-check*[*i*] of the deepest variable with which *v*[*i*] performed a consistency check. If no value can be found in *current-domain*[*i*] that is consistent with the past variables, BJ jumps back to *v*[*h*], where *h* = *max-check*[*i*]. That is, *v*[*h*] is the deepest variable that precludes a candidate value for the current variable, and if *v*[*h*] is reinstantiated a consistent value may be found for *v*[*i*]. If on jumping back to *v*[*h*] there are no remaining values to be tried in *current-domain*[*h*], BJ then chronologically backtracks. Since *v*[*h*] must have passed consistency checks with all past variables, *max-check*[*h*] will be equal to *h*−1, and when BJ jumps back from *v*[*h*] it will actually step back to *v*[*h*−1]. Therefore, we might say that BJ is an algorithm that jumps and steps back. The function below, *bj-label*, corresponds to the labeling function for BJ. *bj-label* requires the global integer array *max-check*. *max-check*[*i*] is initialized to zero for all *i*.

```
 1  FUNCTION bj-label (i,consistent): INTEGER
 2  BEGIN
 3    consistent ← false;
 4    FOR v[i] ← EACH ELEMENT OF current-domain[i] WHILE not consistent
 5    DO BEGIN
 6      consistent ← true;
 7      FOR h ← 1 TO i−1 WHILE consistent
 8      DO BEGIN
 9        consistent ← check(i,h);
10        max-check[i] ← max(max-check[i],h)
```

```
11          END;
12        IF not consistent
13        THEN current-domain[i] ← remove(v[i],current-domain[i])
14        END;
15     IF consistent THEN return(i+1) ELSE return(i)
16  END;
```

Function *bj-label* can be viewed as a modified version of *bt-label*. Whenever checking takes place between *v*[*i*] and *v*[*h*] the array element *max-check*[*i*] is updated (line 10). *bt-label* and *bj-label* differ only in this respect. The function below, *bj-unlabel*, performs the "jumping/stepping" back from the current variable *v*[*i*] to the past variable *v*[*h*].

```
 1  FUNCTION bj-unlabel (i,consistent): INTEGER
 2  BEGIN
 3     h ← max-check[i];
 4     FOR j ← h+1 TO i
 5     DO BEGIN
 6        max-check[j] ← 0;
 7        current-domain[j] ← domain[j]
 8        END;
 9     current-domain[h] ← remove(v[h],current-domain[h]);
10     consistent ← current-domain[h] ≠ nil;
11     return(h)
12  END;
```

In line 3, *h* is selected as the backtracking point. In the FOR loop (lines 4–8) the variables *v*[*j*] are reset (for $h < j \leq i$). By "reset" we mean that *current-domain*[*j*] is reset to *domain*[*j*] and *max-check*[*j*] is reset to zero. Therefore, we are again assured that the *current-domain* of the future variables are equal to their respective *domain*. Lines 9–11 correspond to lines 5–7 in function *bt-unlabel*.

Figure 3 demonstrates the behavior of BJ. The current variable *v*[5] has failed consistency checks with *v*[3] and *v*[1], and it is assumed that there are no values remaining in *current-domain*[5]. BJ then jumps back to *v*[3]. *v*[3] has passed all consistency checks with the variables in its past, namely *v*[1] and *v*[2], and *max-check*[3] = 2. If *v*[3] has no remaining values in *current-domain*[3], BJ "steps" back to *v*[2]. BJ then proceeds to reenumerate the search tree rooted on *v*[3].

In some respects it is important to note what BJ does not do, and we can do this by being clear about the semantics of *max-check*[*i*]. *max-check*[*i*] is not the deepest variable that some trial instantiation of *v*[*i*] was in conflict with, but is the deepest variable that *v*[*i*] checked against. It is only when BJ moves forward from *v*[*h*] to *v*[*i*] and fails to find an instantiation for *v*[*i*] that *max-check*[*i*] is surely the deepest variable that *v*[*i*] failed against. If *max-check*[*i*] was always the deepest variable that precluded some candidate value from *current-domain*[*i*] (and we could do this by updating *max-check*[*i*] only when a consistency check fails) we would have an incomplete algorithm. Assume we change the semantics of *max-check* accordingly, and change its name to *max-fail*, such that *max-fail*[*i*] is the deepest variable that was in conflict with *v*[*i*]. Assume that *v*[*i*] was in conflict with *v*[*g*] and *v*[*h*], where $g < h$. *max-fail*[*i*] will then be *h*. Assume that we then jump back to *v*[*h*], and *v*[*h*] has experienced conflicts with *v*[*f*], where $f < g$. If there are no values in *current-domain*[*h*] our "buggy" version of BJ would jump back to *v*[*f*]. The algorithm has jumped back too far; it should have jumped back to *v*[*g*]. Therefore BJ is conservative but safe, in that it

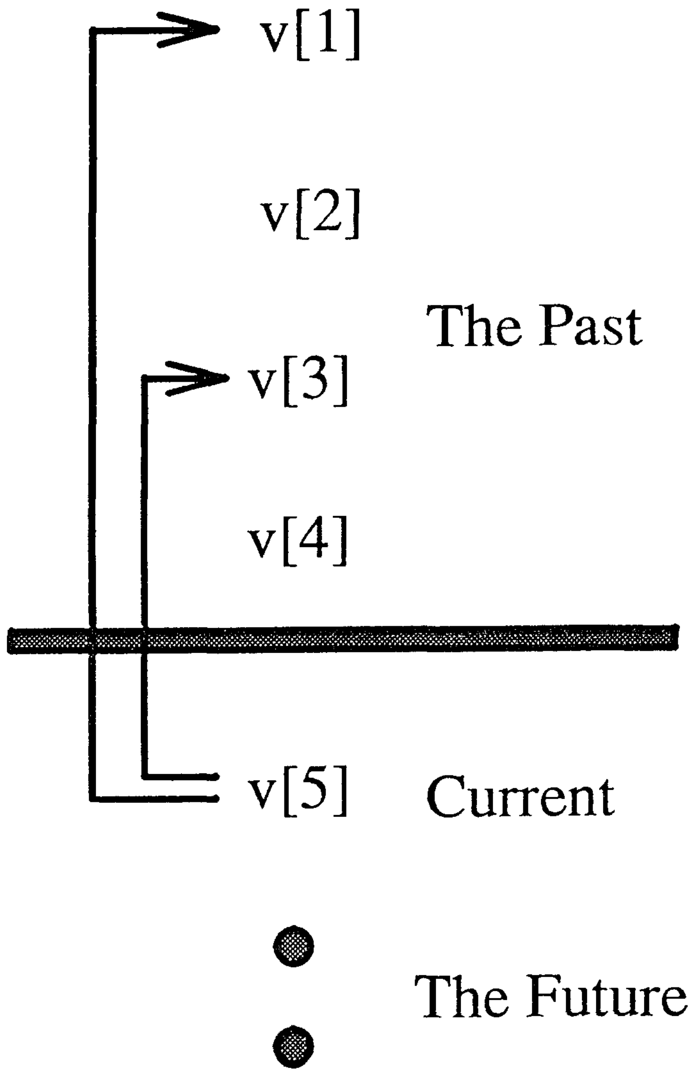

FIGURE 3. A backward-checking scenario.

jumps and then steps back.[4] We might then say that when jumping back BJ is directed by consistency checks that have been performed rather than consistency checks which failed. If BJ was able to remember the set of variables that were in conflict with $v[i]$ it should then be able to make a series of jumps back.

3.3. Conflict-Directed Backjumping (CBJ)

Where BJ steps back from $v[h]$ after jumping back from $v[i]$, the conflict-directed backjumper (CBJ) continues to jump across conflicts which involve both $v[h]$ and $v[i]$. CBJ achieves this by recording the set of past variables that failed consistency checks with the

[4]It may be of interest to note that in Gaschnig's thesis (1979) BJ was presented "without formal proof" (p. 170) and further, that suggested future work was to "prove that backmark and backjump are valid algorithms" (p. 239).

current variable (and we refer to this as a "conflict set" as in Dechter 1990). If no consistent instantiation can be found for *v*[*i*], we then jump back to the deepest variable, *v*[*h*], that conflicted with *v*[*i*]. If on jumping back to *v*[*h*] we discover that there are no more values to be tried in *current-domain*[*h*] we then jump back to *v*[*g*], where *v*[*g*] is the deepest variable that was in conflict with either *v*[*i*] or *v*[*h*].

CBJ maintains a conflict set *conf-set*[*i*] for each variable, where the array *conf-set* is declared globally. Initially each element of *conf-set*[*i*] is set to be {0}. When a consistency check fails between *v*[*i*] and *v*[*h*], *h* is added to the set *conf-set*[*i*]. Therefore, *conf-set*[*i*] is the subset of the past variables in conflict with *v*[*i*]. If there are no remaining values to be tried in *current-domain*[*i*], CBJ jumps back to the deepest variable *v*[*h*], where $h \in$ *conf-set*[*i*] (that is $h \leftarrow$ *max-list*(*conf-set*[*i*]), where the function *max-list* delivers the largest integer in a set of integers). When jumping back from *v*[*i*] to *v*[*h*] the information in *conf-set*[*i*] is carried upward to *v*[*h*]. The array element *conf-set*[*h*] becomes *conf-set*[*h*] $\cup$ *conf-set*[*i*] $-$ *h*, the set of variables in conflict with *v*[*h*] and *v*[*i*]. Therefore when further backtracking takes place from *v*[*h*], CBJ jumps back to *v*[*g*], where *v*[*g*] is the deepest variable in conflict with either *v*[*h*] or *v*[*i*]. It might be said that CBJ performs a primitive style of "learning while searching" (Dechter 1990), and that the current search knowledge exists within *conf-set*[*i*] and *current-domain*[*i*]. *conf-set*[*i*] can be considered as a naive explanation of why values have been removed from *current-domain*[*i*], and is similar to the *causelist* of Rosiers and Bruynooghe (1987).

```
 1  FUNCTION cbj-label (i,consistent): INTEGER
 2  BEGIN
 3    consistent ← false;
 4    FOR v[i] ← EACH ELEMENT OF current-domain[i] WHILE not consistent
 5    DO BEGIN
 6      consistent ← true;
 7      FOR h ← 1 TO i−1 WHILE consistent
 8      DO consistent ← check(i,h);
 9      IF not consistent
10      THEN BEGIN
11        pushnew(h−1,conf-set[i]);
12        current-domain[i] ← remove(v[i],current-domain[i])
13        END
14      END;
15    IF consistent THEN return(i+1) ELSE return(i)
16  END;
```

Function *cbj-label* is very similar to *bj-label*. In *bj-label* the array element *max-check*[*i*] is maintained unconditionally (line 10 of *bj-label*), whereas in *cbj-label* the array element *conf-set*[*i*] is maintained conditionally. Only when a conflict has been detected between *v*[*i*] and *v*[*h*] is *h* added to the set *conf-set*[*i*] (in line 11 the call *pushnew*($h-1$,*conf-set*[*i*]) adds $h-1$ to the set *conf-set*[*i*] if $h-1$ is not already a member of *conf-set*[*i*]).[5]

```
 1  FUNCTION cbj-unlabel (i,consistent): INTEGER
 2  BEGIN
 3    h ← max-list(conf-set[i]);
 4    conf-sett[h] ← remove(h,union(conf-set[h],conf-set[i]));
```

[5]Note: we have to decrement *h* in line 11. This is so because on termination of the FOR loop of line 7, *h* will have a value one greater than during the last execution of the statement of line 8. This is explained more fully in Appendix A.

```
 5    FOR j ← h+1 TO i
 6    DO BEGIN
 7      conf-set[j] ← {0};
 8      current-domain[j] ← domain[j]
 9      END;
10    current-domain[h] ← remove(v[h],current-domain[h]);
11    consistent ← current-domain[h] ≠ nil;
12    return(h)
13    END;
```

Again, function *cbj-unlabel* is similar to *bj-unlabel*. The backtracking point *h* (line 3) is the largest value in the set *conf-set*[*i*], whereas in *bj-unlabel* is *max-check*[*i*]. However, there is no analogue to line 4 within *bj-unlabel*. *bj-unlabel* makes no attempt to pass search knowledge upward through the search tree. If *cbj-label* is modified such that *h* is added to *conf-set*[*i*] unconditionally (move line 11 to position 8.1) CBJ will behave as BJ. An informal proof of the completeness of CBJ, using induction, has been given by Tsang (1992).

In Fig. 3 a call to *cbj-unlabel*(5,*consistent*) would be made, with *conf-set*[5] = {0,1,3}. Variables *v*[4] and *v*[3] would be uninstantiated, *conf-set*[3] would become {0,1}, *consistent* would be *false*, and *cbj-unlabel* would deliver as a result the value 3. Procedure *bcssp* would then make a call to *cbj-unlabel*(3,*consistent*), and *v*[2] and *v*[1] would be uninstantiated. The function call would terminate with *consistent* set to *true*, *conf-set*[1] = {0}, and would deliver a result of 1.

CBJ has many features in common with Dechter's graph-based backjumping algorithm GBJ (Dechter 1990). GBJ exploits the topology of the constraint graph when backtracking. When GBJ reaches a dead on *v*[*i*] it jumps back to the deepest variable among those connected to *v*[*i*] in the constraint graph, namely *v*[*h*], and if there are no values remaining to be tried for *v*[*h*] GBJ jumps back to *v*[*g*] where *v*[*g*] is the deepest variable connected to either *v*[*i*] or *v*[*h*]. GBJ computes for each variable the set *parents*[*i*], where *parents*[*i*] is the set of variables in *v*[*i*]'s past that are connected to *v*[*i*]. When no instantiation can be found for *v*[*i*] GBJ updates the global variable *P* (called the parent set), such that $P \leftarrow P \cup parents[i] - i$, and jumps back to *v*[*h*], where $h \leftarrow max\text{-}list(P)$. The contents of *P* are then carried forward by GBJ. *P* has an alternative interpretation, namely *P* is the superset of variables that have been involved in conflicts experienced by the search process. If the search process reaches a dead-end a safe action is then to jump back to the deepest variable in *P*. Clearly, from our discussion on BJ, it can be seen that if *P* was dispensed with, or was reset whenever a successful forward move was made, we would again have an incomplete algorithm.

3.4. Backmarking (BM)

The backmarking algorithm (BM) of Gaschnig (1977) attempts to minimize the execution of redundant consistency checks within a chronological backtracking algorithm. BM recognizes two situations where redundant checks can be avoided. The first situation is when the current variable *V*[*i*] is about to be reinstantiated with a value *k*, this instantiation previously failed consistency checking with some past variable *v*[*h*], and it is known that *v*[*h*] still holds that conflicting value. Therefore the consistency check will fail again, and BM need not consider that instantiation of *v*[*i*]. The second situation is when BM reinstantiates *v*[*i*] with the value *k*, and it is known that earlier on in the search process consistency checking succeeded between that instantiation of *v*[*i*] and *v*[*h*]. It is known that *v*[*h*] has not changed value; therefore that check will again succeed. Therefore BM does not perform the check between *v*[*h*] and *v*[*i*], and neither does it perform consistency checks between *v*[*i*] and any variable in the past of *v*[*h*].

BM employs two arrays in order to achieve these savings, namely *mcl* (the maximum checking level) and *mbl* (the minimum backup level). *mcl* is an $n \times m$ integer array, where m is the size of the largest domain of the n variables, and *mbl* is a one dimensional integer array of n elements. Initially all elements of *mcl* and *mbl* are set to 0. The array element *mcl*[*i*, *k*] is very similar to the array element *max-check*[*i*] in BJ. That is, *max-check*[*i*] is the deepest variable that *v*[*i*] checked against, whereas *mcl*[*i*, *k*] is the deepest variable that the instantiation *v*[*i*] $\leftarrow$ *k* checked against. Therefore, *mcl*[*i*, *k*] is a finer grained version of *max-check*[*i*]. The array element *mbl*[*i*] records the shallowest past variable that has changed value since *v*[*i*] was the current variable. Therefore, let *h* be *mbl*[*i*]. BM is aware that all variables in the past of *v*[*h*] have not changed value since BM last visited *v*[*i*].

The arrays *mcl* and *mbl* are exploited as follows. Assume BM attempts the instantiation *v*[*i*] $\leftarrow$ *k*. If *mcl*[*i*, *k*] < *mbl*[*i*] consistency checking must have failed between *v*[*i*] $\leftarrow$ *k* and the variable *v*[*h*], where *h* = *mcl*[*i*, *k*]. The variable *v*[*h*] has not reinstantiated with a value, and consistency checking will again fail. Therefore the instantiation *v*[*i*] $\leftarrow$ *k* need not be considered. This is referred to as "type (a) saving" in Nadel (1989). When *mcl*[*i*, *k*] $\geq$ *mbl*[*i*], the instantiation *v*[*i*] $\leftarrow$ *k* must have passed consistency checking with the variables *v*[*h*], for all *h*, where *h* < *mbl*[*i*]. Since these variables have not been re-instantiated these checks will continue to succeed. Therefore consistency checking need only be performed against variables *v*[*h*], for all *h*, where *mbl*[*i*] $\leq$ *h* < *i*. This is referred to as a "type (b) saving" in Nadel (1989). The functions below, *bm-label* and *bm-unlabel*, describe an explicit form of backmarking.

```
 1  FUNCTION bm-label (i,consistent): INTEGER
 2  BEGIN
 3    consistent ← false;
 4    FOR k ← EACH ELEMENT OF current-domain[i] WHILE not consistent
 5    DO BEGIN
 6      consistent ← mcl[i,k] ≥ mbl[i];
 7      FOR h ← mbl[i] TO i−1 WHILE consistent
 8      DO BEGIN
 9        v[i] ← k;
10        consistent ← check(i,h);
11        mcl[i,k] ← h
12        END;
13      IF not consistent
14      THEN current-domain[i] ← remove(v[i],current-domain[i])
15      END;
16    IF consistent THEN return(i+1) ELSE return(i)
17  END;
```

The type (a) savings are achieved via line 6 above. That is, if *mcl*[*i*, *k*] < *mbl*[*i*] the FOR loop (lines 7 to 12) is not executed. Type (b) savings are achieved via the lower bound *mbl*[*i*] of the FOR loop in line 7. Again, *bm-label* may be compared with *bj-label*. In *bj-label* the array element *max-check*[*i*] is maintained unconditionally (line 10 of *bj-label*), and in *bm-label* the array element *mcl*[*i*, *k*] is maintained conditionally (line 11 above). *bj-label* records only the deepest variable that *v*[*i*] checked against, whereas *bm-label* records, for each instantiation *v*[*i*] $\leftarrow$ *k*, the deepest variable that that instantiation checked against.

Therefore it appears that the information in *mcl* can be exploited to allow BJ within BM (and this will be shown in the next section).

```
 1  FUNCTION bm-unlabel (i,consistent): INTEGER
 2  BEGIN
 3    h ← i−1;
 4    current-domain[i] ← domain[i];
 5    mbl[i] ← h;
 6    FOR j ← h+1 TO n DO mbl[j] ← min(mbl[j],h);
 7    current-domain[h] ← remove (v[h],current-domain[h]);
 8    consistent ← current-domain[h] ≠ nil;
 9    return(h);
10  END;
```

Lines 5 and 6 maintain the backtracking information within *mbl*, and if we remove lines 5 and 6, *bm-unlabel* becomes *bt-unlabel*. It is worth noting that BM is dependent upon a static order of instantiation. If the order of instantiation was allowed to change during the search process this would result in a corruption of the search knowledge within the arrays *mcl* and *mbl*. Therefore BM, and any hybrids of BM, cannot exploit heuristics that examine future variables during the search process.

3.5. Backmarking and Backjumping (BMJ)

From the discussion on BM it appears that BJ can be incorporated within BM, resulting in the hybrid BMJ. It is anticipated that BMJ will enjoy the advantages of BM and BJ, namely, avoiding redundant consistency checks while reducing the number of nodes visited within the search tree. That is, BMJ should make the type (a) and (b) savings described earlier, while being able to jump back to the source of conflicts. We can do this by modifying *bm-label* such that it maintains the information required by *bj-unlabel* (namely *max-check*[*i*]), and by modifying *bj-unlabel* such that it maintains the information required by *bm-label* (namely *mbl*[*i*]). BMJ is then defined by the functions, *bmj-label* and *bmj-unlabel*.

```
 1    FUNCTION bmj-label (i,consistent): INTEGER
 2    BEGIN
 3      consistent ← false;
 4      FOR k ← EACH ELEMENT OF current-domain[i] WHILE not consistent
 5      DO BEGIN
 6        consistent ← mcl[i,k] ≥ mbl[i];
 7        FOR h ← mbl[i] TO i−1 WHILE consistent
 8        DO BEGIN
 9          v[i] ← k;
10          consistent ← check(i,h);
11          mcl[i,k] ← h
12          END;
12.1      max-check[i] ← max(max-check[i],mcl[i,k]);
13        IF not consistent
14        THEN current-domain[i] ← remove(v[i],current-domain[i])
15        END;
16      IF consistent THEN return(i+1) ELSE return(i)
17    END;
```

By adding line 12.1 to *bm-label* we get *bmj-label*. This addition maintains the information required for backjumping, and corresponds to line 10 in *bj-label* (replacing h with $mcl[i, k]$). On reaching line 12.1 one of three states will hold, namely (i) $mcl[i,k] < mbl[i]$ and $consistent = false$, (ii) $mcl[i,k] \geq mbl[i]$ and $consistent = true$, or (iii) $mcl[i,k] \geq mbl[i]$ and $consistent = false$. In state (i) a type (a) saving has been made by *bmj-label* and $mcl[i,k]$ is the variable with which the instantiation $v[i] \leftarrow k$ failed consistency checking in some previous call to *bmj-label*. In state (ii) a type (b) saving has been made by *bmj-label*, and $mcl[i,k] = i-1$. In state (iii) a type (b) saving has been made by *bmj-label*, and $mcl[i,k]$ is the variable with which the instantiation $v[i] \leftarrow k$ failed consistency checking in this call to *bmj-label*. Therefore in states (i) and (iii) consistency checking failed, but in state (i) this failure was detected in a previous call to *bmj-label*.

```
1    FUNCTION bmj-unlabel (i,consistent): INTEGER
2    BEGIN
3      h ← max-check[i];
3.1    mbl[i] ← h;
3.2    FOR j ← h+1 TO n DO mbl[j] ← min(mbl[j],h);
4      FOR j ← h+1 TO i
5      DO BEGIN
6        max-check[j] ← 0;
7        current-domain[j] ← domain[j]
8        END;
9      current-domain[h] ← remove(v[h],current-domain[h]);
10     consistent ← current-domain[h] ≠ nil;
11     return(h)
12   END;
```

By adding lines 3.1 and 3.2 to *bj-unlabel* we get *bmj-unlabel*. This addition maintains the information required by backmarking, and corresponds to lines 5 and 6 in *bm-unlabel*. As previously noted we might anticipate that BMJ will enjoy the advantages of both BM and BJ. *Caveat actor*: a careful study of *bmj-unlabel* reveals a scenario where BMJ might perform worse than BM. Assume BMJ jumps from $v[i]$, over $v[h]$, to $v[g]$, and does so when $mbl[h] < g$. When $v[h]$ again becomes the current variable consistency checks will be repeated between $v[h]$ and the variable $v[f]$, for all f, where $mbl[h] \leq f < g$. Therefore we can only be sure that BMJ enjoys *some* of the advantages of BM.

3.6. Backmarking and Conflict-Directed Backjumping (BM-CBJ)

The hybrid of BM and CBJ (BM-CBJ) can be realized by again modifying *bm-label* such that it maintains the information required by *cbj-unlabel* (namely *conf-set*[i]), and by modifying *cbj-unlabel* such that it maintains the information required by *bm-label* (namely $mbl[i]$). We should expect that BM-CBJ will be able to make the type (a) and (b) savings of BM, while being able to make the multiple jumps of CBJ. BM-CBJ is then defined by the following functions, *bm-cbj-label* and *bm-cbj-unlabel*.

```
1    FUNCTION bm-cbj-label (i,consistent): INTEGER
2    BEGIN
3      consistent ← false;
4      FOR k ← EACH ELEMENT OF current-domain[i] WHILE not consistent
5      DO BEGIN
```

```
6        consistent ← mcl[i,k] ≥ mbl[i]
7        FOR h ← mbl[i] TO i−1 WHILE consistent
8        DO BEGIN
9          v[i] ← k;
10         consistent ← check(i,h);
11         mcl[i,k] ← h
12         END;
13       IF not consistent
14       THEN BEGIN
14.1         pushnew(mcl[i,k],conf-set[i]);
14.2         current-domain[i] ← remove(v[i],current-domain[i])
14.3         END
15       END;
16     IF consistent THEN return(i+1) ELSE return(i)
17   END;
```

Line 14 of *bm-label* has been modified (lines 14.1 to 14.3 above) to give *bm-cbj-label*. Lines 14.1 and 14.2 correspond to lines 11 and 12 in *cbj-label* (replacing $h-1$ with *mcl*[*i*, *k*]) and maintains the information required by conflict-directed backjumping. On reaching line 14.1 it has been discovered that the instantiation *v*[*i*] ← *k* was inconsistent. This may have been discovered due to *mcl*[*i*, *k*] being less than *mbl*[*i*] (line 6), or due to the failure of a consistency check at line 10. In both cases *mcl*[*i*, *k*] will be the variable with which consistency checking has failed, either in this call to *bm-cbj-label* (line 10) or in some previous call to *bm-cbj-label* (line 6). Therefore *mcl*[*i*, *k*] is added to the set *conf-set*[*i*].

```
1    FUNCTION bm-cbj-unlabel.(i,consistent): INTEGER
2    BEGIN
3      h ← max-list(conf-set[i]);
4      conf-set[h] ← remove(h,union(conf-set[h],conf-set[i]));
4.1    mbl[i] ← h;
4.2    FOR j ← h+1 TO n DO mbl[j] ← min(mbl[j],h);
5      FOR j ← h+1 TO i
6      DO BEGIN
7        conf-set[j] ← {0};
8        current-domain[j] ← domain[j];
9        END;
10     current-domain[h] ← remove(v[h],current-domain[h]);
11     consistent ← current-domain[h] ≠ nil;
12     return(h)
13   END;
```

By adding lines 4.1 and 4.2 to *cbj-unlabel* we get *bm-cbj-unlabel*. These additions correspond to lines 5 and 6 in *bm-unlabel*. BM-CBJ will be prone to the same weaknesses as BMJ. That is, when BM-CBJ jumps from *v*[*i*], over *v*[*h*], to *v*[g], when $mbl[h] < g$, redundant consistency checking may take place between *v*[*h*] and *v*[*f*] (for all *f*, where $mbl[h] \leq f < g$). Indeed, we might expect BM-CBJ to perform worse than BMJ when BMJ performs worse than BM. This is so because BJ "jumps" then "steps" back, whereas CBJ can "jump" and continue jumping. When jumping back the BM hybrids will be prone to the above weakness, and BM-CBJ will tend to jump more frequently than BMJ.

3.7. Explicit Forward Checking (FC)

The forward-checking algorithm (FC) of Haralick and Elliott (1980) is a "looking ahead" scheme. When the search process makes a trial instantiation of a variable it looks ahead toward the future variables, and removes from the *current-domain* of those variables all values that are incompatible with the trial instantiation. Therefore, when FC again moves forward, and considers some new variable, we can be sure that all values in its *current-domain* are consistant with the past variables. If this "looking ahead" results in the annihilation of the *current-domain* of some future variable (and Nadel (1989) refers to this as a "domain wipe out") a new value is tried for the current variable, and if no values remain to be tried FC backtracks chronologically. The goal of forward checking is to "fail early" by detecting inconsistencies within the search tree as early as possible, thus saving the exploration of fruitless alternatives. Forward checking performs more work per node than the algorithms presented so far, but attempts to visit as few nodes as possible. It is hope that this results in a net saving in consistency checks performed during the search process.

Forward checking may be made explicit as follows. When the instantiation $v[i] \leftarrow k$ is attempted *current-domain*[j] is filtered, for all j, where $i < j \leq n$. The effects of filtering, from $v[i]$ to $v[j]$, are recorded explicitly within the (global) array elements *reductions*[j], *future-fc*[i], and *past-fc*[j]. The array element *reductions*[j] is a sequence of sequences, and is initialized to *nil*. Let *reduction* $\in$ *reductions*[j]. *reduction* is a sequence of values that are disallowed in *current-domain*[j] due to the instantiation of one of the past variables. The array element *future-fc*[i] is a set (and is treated as a stack, initialized to *nil*) representing the subset of the future variables that $v[i]$ checks against. Let $j \in$ *future-fc*[i]. This is interpreted as follows: the current instantiation of $v[i]$ forward checks against the future variable $v[j]$ and disallows a sequence of values in *current-domain*[j]. The array element *past-fc*[j] is a set (and is treated as a stack, initialized to $\{0\}$) representing the subset of the past variables that check against $v[j]$. Let $i \in$ *past-fc*[j]. This is interpreted as follows: the current instantiation of $v[i]$ forward checks against $v[j]$ disallowing a sequence of values from *current-domain*[j].[6]

The function below, *check-forward*, is called when the variable $v[i]$ is instantiated with a value. It removes all values from *current-domain*[j] which are inconsistent with the current instantiation of $v[i]$, where $i < j$. It returns a result of *true* if there are values remaining in *current-domain*[j], otherwise it delivers *false*.

```
FUNCTION check-forward(i,j): BOOLEAN
BEGIN
  reduction ← nil;
  FOR v[j] ← EACH ELEMENT OF current-domain[j]
  DO IF not check(i,j)
    THEN push(v[j],reduction);
  IF reduction ≠ nil
  THEN BEGIN
      current-domain[j] ← set-difference(current-domain[j],reduction);
      push(reduction,reductions[j]);
      push(j,future-fc[i]);
      push(i,past-fc[j])
```

[6]The array element past-fc[j] will be exploited when combining forward checking with BJ and CBJ. past-fc[j] is used in conflict detection and subsequent backtracking. However, past-fc[j] is not used by the chronological backtracker FC, and is introduced at this point only for convenience.

```
13        END;
14    return(current-domain[j] ≠ nil)
15  END;
```

On termination of the FOR loop the local variable *reduction* will contain the sequence of values in *current-domain*[*j*] that are inconsistent with respect to *v*[*i*]. In lines 9 to 12 a record is maintained of the effects of forward checking. The procedure below, *undo-reductions*, is called whenever the variable *v*[*i*] is uninstantiated. The procedure undoes all effects of forward checking from *v*[*i*].

```
1   PROCEDURE undo-reductions(i)
2   BEGIN
3     FOR j ← EACH ELEMENT OF future-fc[i]
4     DO BEGIN
5       reduction ← pop(reductions[j]);
6       current-domain[j] ← union(current-domain[j],reduction);
7       pop(past-fc[j])
8       END;
9     future-fc[i] ← nil
10  END;
```

The statement in line 5, *pop*(*reduction*[*j*]), removes the most recent *reduction* from *reductions*[*j*], and line 7 removes the backward reference from *v*[*j*] to *v*[*i*]. In line 9 *future-fc*[*i*] is set to *nil* because *v*[*i*] no longer forward checks against any variable.

The procedure below, *update-current-domain*, recomputes *current-domain*[*i*] to be *domain*[*i*] less all values disallowed by forward checking (namely *reductions*[*i*]). This procedure is called whenever *v*[*i*] is the current variable and there are no values remaining in *current-domain*[*i*].[7]

```
1  PROCEDURE updated-current-domain(i)
2  BEGIN
3    current-domain[i] ← domain[i];
4    FOR reduction ← EACH ELEMENT OF reductions[i]
5    DO current-domain[i] ← set-difference(current-domain[i],reduction)
6  END;
```

The main body explicit forward checking can now be presented. The function below, *fc-label*, attempts to instantiate the current variable *v*[*i*]. When an instantiation is attempted, the *current-domain* of the future variables are filtered. Whenever a *current-domain*[*j*] becomes empty, the instantiation of *v*[*i*] is retracted and the effects of domain filtering from *v*[*i*] are undone.

```
1  FUNCTION fc-label(i,consistent): INTEGER
2  BEGIN
3    consistent ← false;
4    FOR v[i] ← EACH ELEMENT OF current-domain[i] WHILE not consistent
5    DO BEGIN
6      consistent ← true;
7      FOR j ← i+1 TO n WHILE consistent
8      DO consistent ← check-forward(i,j);
```

[7]A call to update-current domain(*i*) is the "FC equivalent" of the statement current-domain[*i*] ← domain[*i*] encountered in the previous "backward"-checking algorithms.

```
 9      IF not consistent
10      THEN BEGIN
11        current-domain[i] ← remove(v[i],current-domain[i]);
12        undo-reductions(i)
13        END
14      END;
15    IF consistent THEN return(i+1) ELSE return(i)
16  END;
```

fc-label should be compared with *bt-label*. These two functions differ in the FOR loop of line 7. *bt-label* varies *h*, from 1 to $i-1$, checking backward against past variables, and *fc-label* varies *j*, from $i+1$ to *n*, checking against the future variables. In line 8 of *bt-label* a call is made to *check*(*i*, *h*), and in line 8 of *fc-label* a call is made to *check-forward*(*i*,*j*). The only real addition is line 12 in *fc-label*, where the effects of forward checking are undone via the call to *undo-reductions*(*i*).

```
1  FUNCTION fc-unlabel(i,consistent): INTEGER
2  BEGIN
3    h ← i−1;
4    undo-reductions(h);
5    update-current-domain[i];
6    current-domain[h] ← remove(v[h],current-domain[h]);
7    consistent ← current-domain[h] ≠ nil
8    return(h);
9  END;
```

In line 4 the effects of forward check from *v*[*h*] are undone, and in line 5 *current-domain*[*i*] is recomputed. *fc-unlabel* should be compared to *bt-unlabel*. If we remove line 4 above, and replace line 5 with the statement *current-domain*[*i*] ← *domain*[*i*], the function *fc-unlabel* becomes *bt-unlabel*.

3.8. Explicit Forward Checking and Backjumping (FC-BJ)

FC is prone to the same vagaries as BT, namely thrashing. There is nothing to prevent FC from chronologically backtracking to a variable that plays no role in the current conflict. Figure 4 demonstrates such a scenario within FC.

In Figure 4. *v*[5] is the current variable, the past variables are above the bold line, *v*[1] forward checks against *v*[5], and *v*[3] forward checks against *v*[6]. In the call to function *fc-label*(5,*consistent*), no value can be found in *current-domain*[5] that is consistent with some value in *current-domain*[6]. The function call *fc-label*(5,*consistent*) terminates with *consistent* set to false, and delivers the integer result 5. *bcssp* then backtracks to *v*[4] via the function call *fc-unlabel*(5,*consistent*). This does not relax *current-domain*[6] or *current-domain*[5]. The search subtree routed on *v*[5] will then be reexplored with identical results. This process will be repeated until *current-domain*[4] is exhausted. FC could have avoided this scenario if it had jumped back to *v*[3], relaxing *current-domain*[6] as a result of the call to *undo-reductions*(3).

This weakness can be addressed by giving FC the "jumping" capability of BJ. The explicit representation of forward checking allows this to be done with relative ease. When *check-forward*(*i*,*j*) delivers a result of *false*, the information in *past-fc*[*i*] and *past-fc*[*j*] can

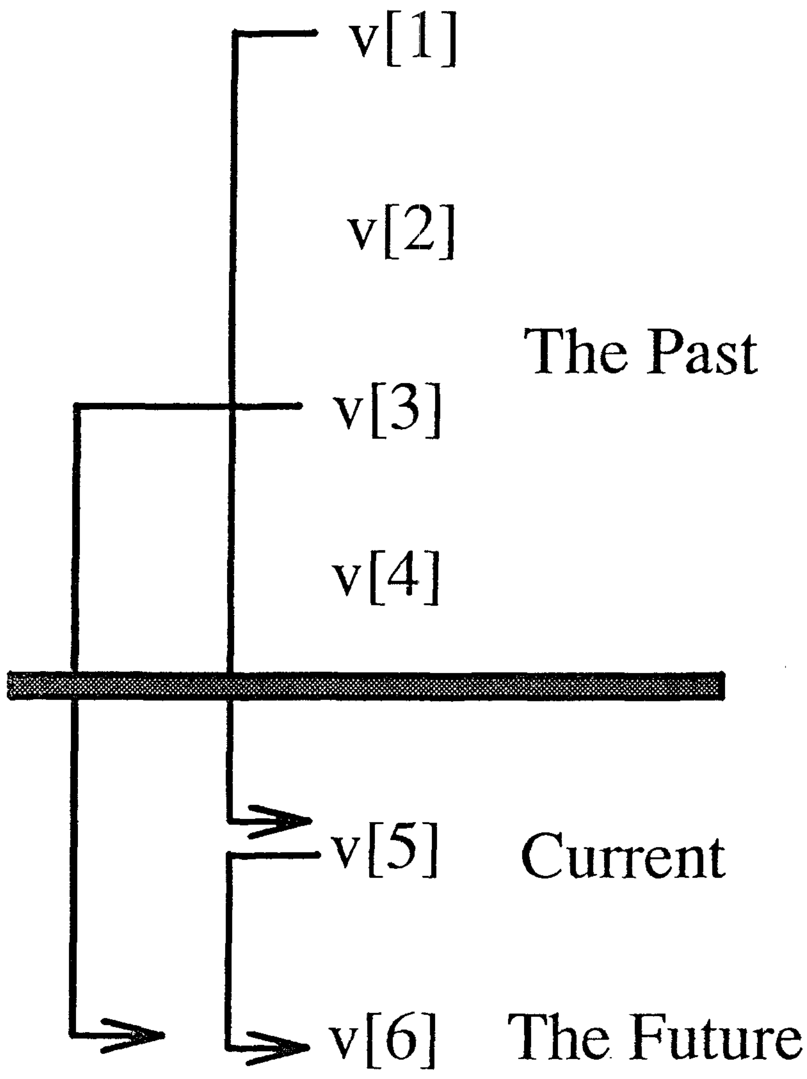

FIGURE 4. A forward-checking scenario.

be analyzed to deliver a backtracking point. FC-BJ is realized by modifying *fc-label* and *bj-unlabel*.[8]

```
1 FUNCTION fc-bj-label(i,consistent): INTEGER
2 BEGIN
3   consistent ← false;
4   FOR v[i] ← EACH ELEMENT OF current-domain[i] WHILE not consistent
5   DO BEGIN
6     consistent ← true;
7     FOR j ← i+1 TO n WHILE consistent
```

[8]As far as I am aware, FC-BJ was first described by Ottestad (1991). Unfortunately that definition is flawed. Ottestad's algorithm is overly "optimistic" when it jumps back, and may prune out solutions. Therefore, it was not complete.

```
8      DO consistent ← check-forward(i,j);
9      IF not consistent
10     THEN BEGIN
11       current-domain[i] ← remove(v[i],current-domain[i]);
12       undo-reductions(i);
12.1     max-check[i] ← max(max-check[i],max-list(past-fc[j−1]))
13       END
13.1   ELSE max-check[i] ← i−1;
14     END;
15   IF consistent THEN return(i+1) ELSE return(i)
16   END;
```

Lines 12.1 and 13.1 maintain the information required for backjumping, namely *max-check*[*i*].[9] On reaching line 12.1 *v*[*i*] forward checked against *v*[*j*−1] and annihilated *current-domain* [*j*−1]. The reason why this happened was due to the variables forward checking against *v*[*j*−1], namely *past-fc*[*j*−1], and the attempted instantiation of *v*[*i*]. On reaching line 12.1, *i* no longer forward checks against *v*[*j*−1]. This is due to the call to *undo-reductions*(*i*) in line 12, and *i* is no longer a member of *past-fc*[*j*−1]. Therefore, in line 12.1 *max-check*[*i*] becomes either the deepest variable that forward checks against *v*[*j*−1] or the previous value of *max-check*[*i*].

To prevent the conflict between *v*[*i*] and *v*[*j*] from recurring we have the following options: (i) relax *current-domain*[*j*] by uninstantiating the deepest variable forward checking against *v*[*j*] (namely *max-check*[*i*]), or (ii) relax *current-domain*[*i*] by uninstantiating the deepest variable forward checking against *v*[*i*] (namely *max-list*(*past-fc*[*i*])). Option (i) may allow us to instantiate *v*[*i*] with the value that was last used, and this may be consistent with some value in the relaxed *current-domain*[*j*]. Option (ii) may allow us to find a new instantiation for *v*[*i*] from the relaxed *current-domain*[*i*] that is consistent with some value in *current-domain*[*j*]. Line 13.1 is in some respects artificial. This forces FC-BJ to behave in a manner similar to BJ, that is, jumping back followed by stepping back. If we jump back to *v*[*i*], and there are no values remaining in *current-domain*[*i*], we then chronologically backtrack to *h* ← *i*−1.

```
1    FUNCTION fc-bj-unlabel(i,consistent): INTEGER
2    BEGIN
3      h ← max(max-check[i],max-list(past-fc[i]));
4      FOR j ← i DOWNTO h+1
5      DO BEGIN
6        max-check[j] ← 0;
6.1      undo-reductions(j);
7        update-current-domain(j)
8        END
8.1    undo-reductions(h);
9      current-domain[h] ← remove(v[h],current-domain[h]);
10     consistent ← current-domain[h] ≠ nil;
11     return(h)
12   END;
```

[9]Again, we have to decrement *j* in line 12.1 because on termination of the FOR loop of line 7 *j* will have a value that is one greater than during the last execution of the statement of line 8. Therefore, if consistent is false at line 9, *j*−1 is the value that caused check-forward to deliver a result of false.

fc-bj-unlabel is realized by modifying *bj-unlabel*. In line 3 we select the backtracking point *h*. When *h* takes the value *max-check*[*i*], this corresponds to backtracking option (i) described above, and if *h* takes the value *max-list(past-fc*[*i*]) this corresponds to option (ii). The FOR loop of line 4 counts downward, from *i* down to $h+1$, so that the effects of forward checking are properly undone. Line 6.1 is an addition, undoing the effects of forward checking, and line 7 now calls *update-current-domain*(*i*) rather than resetting *current-domain*.

It is expected that FC-BJ will enjoy the advantages of FC and BJ, resulting in a further reduction in nodes visited, leading to a further reduction in consistency checks performed during the search process. However, FC-BJ will still exhibit the BJ characteristic of jumping then stepping back. In Fig. 4 FC-BJ would jump from *v*[5] to *v*[3] and would then proceed to step back to *v*[2].

3.9. Explicit Forward Checking and Conflict-Directed Backjumping (FC-CBJ)

Incorporating conflict-directed backjumping into forward checking is now trivial.[10]

```
1     FUNCTION fc-cbj-label(i,consistent): INTEGER
2     BEGIN
3       consistent ← false;
4       FOR v[i] ← EACH ELEMENT OF current-domain[i] WHILE not consistent
5       DO BEGIN
6         consistent ← true;
7         FOR j ← i+1 TO n WHILE consistent
8         DO consistent ← check-forward(i,j);
9         IF not consistent
10        THEN BEGIN
11          current-domain[i] ← remove(v[i],current-domain[i]);
12          undo-reductions(i);
12.1        conf-set[i] ← union(conf-set[i],past-fc[j−1])
13          END
14        END;
15      IF consistent THEN return(i+1) ELSE return(i)
16    END;
```

By the addition of the line 12.1 to *fc-label* we get *fc-cbj-label*, where line 12.1 maintains the information required for conflict-directed backjumping, namely *conf-set*[*i*].[11] On reaching line 12.1 a conflict has been detected between the instantiation of *v*[*i*] and *current-domain*[$j-1$]. Due to the variables forward checking against *current-domain*[$j-1$] (namely *past-fc*[$j-1$]) the instantiation of *v*[*i*] annihilated *current-domain*[$j-1$]. Therefore *past-fc*[$j-1$] is added to *conf-set*[*i*]. Again, due to the call to *undo-reductions*(*i*) in line 12, *v*[*i*] no longer forward checks against *current-domain*[$j-1$].

```
1   FUNCTION fc-cbj-unlabel(i,consistent): INTEGER
2   BEGIN
3     h ← max(max-list(conf-set[i],max-list(past-fc[i]));
```

[10]In fact it is a more "natural" algorithm than FC-BJ. Function *fc-bj-label* had to be engineered to prevent conflict-directed backjumping and force simple backjumping. It should come as no surprise therefore that FC-CBJ was developed before FC-BJ.

[11]Again we assume the loop variable J is available to line 12.1 and that the value $j-1$ caused check-forward to deliver false.

```
4     conf-set[h] ← remove(h,union(conf-set[h],union(conf-set[i],past-fc[i])));
5     FOR j ← i DOWNTO h+1
6     DO BEGIN
7        conf-set[j] ← {0};
7.1      undo-reductions(j);
8        update-current-domain(j)
9        END;
9.1   undo-reductions(h);
10    current-domain[h] ← remove(v[h],current-domain[h]);
11    consistent ← current-domain[h] ≠ nil
12    return(h)
13  END;
```

fc-cbj-label is realized by modifying *cbj-unlabel* (lines 3, 4, 5, and 8 have been changed, and lines 7.1 and 9.1 have been added). Line 3 compares closely with line 3 of *fc-bj-unlabel*. In order to resolve the conflict involving $v[i]$ we jump back to $v[h]$ where (i) h is the closest past variable that forward checks against some variable in the future of $v[i]$ (namely *max-list*(*conf-set*[i])), or (ii) h is the closest past variable that forward checks against $v[i]$ (namely *max-list*(*past-fc*[i])). In line 4 the conflict set involving $v[h]$ is updated such that it includes (a) the variables in conflict with $v[h]$, and (b) the variables in conflict with $v[i]$, and (c) the variables forward checking against $v[i]$. In Fig. 4 FC-CBJ would backtrack from $v[5]$ to $v[3]$, and then from $v[3]$ to $v[1]$.

4. THE EXPERIMENTS

The experiments were performed over a single problem, namely the ZEBRA, described below. This problem was chosen for a number of reasons. First, the problem is representative of real world design problems (such as in Voss *et al.* 1990), and problems that exist within the scheduling domain (such as in Burke and Prosser 1991; Prosser 1989, 1990). Second, the problem is nontrivial, involving 25 variables and 122 constraints. Third, by permuting the order of instantiation we get significantly different search problems (Freuder 1982). Therefore the ZEBRA problem allows us to choose from potentially 25! different problems. As will be seen, this has allowed us to generate a range of problems, from easy (taking hundreds of consistency checks) through to difficult (taking in excess of 100 million consistency checks).[12]

Finally, the order of instantiation has a number of measurable properties, namely bandwidth (Zabih 1990; Monien and Sudborough 1980), width, and induced width (Dechter 1992). Given the constraint matrix C, and an order of instantiation d, then the bandwidth of a variable $v[i]$ is the maximum value of $|i-j|$, for all j, where $1 \leq j \leq n$ and $C[i,j] \neq nil$. That is, the bandwidth of $v[i]$ is the maximum distance between $v[i]$ and its adjacent predecessors. The bandwidth of the constraint graph G under the ordering d is then the maximum of the bandwidths of the variables, and will be written as $B(d)$. The induced width of G under the ordering d, $W^*(d)$, is a measure taken from the *induced graph*. That is, by recursively connecting any two parents sharing a common successor we induce a new constraint. The width of a variable is the number of adjacent predecessors of that variable, and the width of an ordering is the maximum width of all variables. $W^*(d)$ is then the width of the induced graph under that ordering (Dechter and Meiri 1989). There-

[12]This is not a new idea. This technique has been exploited by Gaschnig (1977) and Dechter (1990).

fore, we investigate (empirically) the effects of these topological parameters on the nine algorithms.

The ZEBRA problem (also described in Dechter 1988, 1990; Smith 1992) is composed of 25 variables. These variables correspond to five houses ($v[1]$ to $v[5]$: Red, Blue, Yellow, Green, Ivory), five brands of cigarettes ($v[6]$ to $v[10]$: Old-Gold, Parliament, Kools, Lucky, Chesterfield), five nationalities ($v[11]$ to $v[15]$: Norwegian, Ukranian, Englishman, Spaniard, Japanese), five pets ($v[16]$ to $v[20]$: Zebra, Dog, Horse, Fox, Snails), and five drinks ($v[21]$ to $v[25]$: Coffee, Tea, Water, Milk, Orange Juice). Each of the variables has a domain of $\langle 1,2,3,4,5 \rangle$, with the exception of the Norwegian $v[1]$ and Milk $v[24]$, as described below. The ZEBRA constraints are described by the following statements:

- Each of the houses is a different color ($C[i,j] \leftarrow \neq$, where $1 \leq i \leq 5$, $1 \leq j \leq 5$, $i \neq j$), inhabited by a single person, ($C[i,j] \leftarrow \neq$, where $11 \leq i \leq 15$, $11 \leq j \leq 15$, $i \neq j$), who smokes a unique brand of cigarette ($C[i,j] \leftarrow \neq$, where $6 \leq i \leq 10$, $6 \leq j \leq 10$, $i \neq j$), has a preferred drink ($C[i,j] \leftarrow \neq$, where $21 \leq i \leq 25$, $21 \leq j \leq 25$, $i \neq j$), and owns a pet ($C[i,j] \leftarrow \neq$, where $16 \leq i \leq 20$, $16 \leq j \leq 20$, $i \neq j$).
- The Englishman lives in the Red house. ($C[13,1] \leftarrow =$ and $C[1,13] \leftarrow =$).
- The Spaniard owns a Dog. ($C[14,17] \leftarrow =$ and $C[17,14] \leftarrow =$).
- Coffee is drunk in the Green house. ($C[21,4] \leftarrow =$ and $C[4,21] \leftarrow =$).
- The Ukranian drinks Tea. ($C[12,22] \leftarrow =$ and $C[22,12] \leftarrow =$).
- The Green house is to the right of the Ivory house.[13] ($C[4,5] \leftarrow >$ and $C[5,4] \leftarrow <$).
- The Old-Gold smoker owns Snails. ($C[6,20] \leftarrow =$ and $C[20,6] \leftarrow =$).
- Kools are smoked in the Yellow house. ($C[8,3] \leftarrow =$ and $C[3,8] \leftarrow =$).
- Milk is drunk in the middle house ($domain[24] = \langle 3 \rangle$).
- The Norwegian lives in the first house on the left ($domain[1] = \langle 1 \rangle$).
- The Chesterfield smoker lives next to the Fox owner. ($C[10,19] \leftarrow$ *next-to* and $C[19,10] \leftarrow$ *next-to*).[14]
- Kools are smoked in the house next to the house where the Horse is kept. ($C[8,18] \leftarrow$ *next-to* and $C[18,8] \leftarrow$ *next-to*).
- The Lucky smoker drinks Orange Juice. ($C[9,25] \leftarrow =$ and $C[25,9] \leftarrow =$).
- The Japanese smokes Parliament. ($C[15,7] \leftarrow =$ and $C[7,15] \leftarrow =$).
- The Norwegian lives next to the Blue house. ($C[11,2] \leftarrow$ *next-to* and $C[2,11] \leftarrow$ *next-to*).

The query is: "Who drinks water, and who owns the Zebra?"[15]

A program was written that randomly searched for 50 instances of the ZEBRA problem at a given bandwidth $B(d)$, such that no two instances represented the same instantiation order. This program was run with $B(d)$ in the range $16 \leq B(d) \leq 24$. In total, 450 problem instances were generated and saved to disk. A program was then developed such that an instantiation order could be read from disk, and the corresponding ZEBRA created. This involved renumbering the above variables and translating the constraint matrix C. In turn, each of the tree search algorithms was applied to each of the problems, and a 6-tuple was captured ⟨I A B W X Y⟩, where I is a unique identifier for that problem instance ($1 \leq I \leq 450$), A was the name of the algorithm ($A \in$ {BT, BJ, CBJ, BM, BMJ, BM-CBJ, FC, FC-

[13]Consequently the constraints C [4,5] and C [5,4] are overwritten.

[14]The relation "X next-to Y" is implemented as $X-Y = \pm 1$.

[15]The above problem definition differs from that in Dechter (1988, 1990) in that "The Green house is to the right of the Ivory house," rather than "immediately to the right of." This relaxes the problem, resulting in 11 possible solutions rather than 1. This feature was exploited when developing the algorithms, in that if two algorithms were given the same instantiation order they should find the same solution. If they did not, then one of the algorithms was clearly incomplete.

TABLE 1. Consistency Checks.

Algorithm	μ	σ	Min	Max
BT	3,858,989	9,616,407	1773	102,267,383
BJ	503,324	1,524,193	358	19,324,081
CBJ	63,212	193,846	339	3,297,304
BM	396,945	1,276,415	401	18,405,514
BMJ	125,474	361,595	300	5,214,608
BM-CBJ	25,470	72,004	297	1,237,283
FC	35,582	71,012	262	802,069
FC-BJ	16,839	29,977	262	280,302
FC-CBJ	10,361	16,383	262	119,767

BJ, FC-CBJ}), B the bandwidth of that problem instance, W the induced width of that ordering, X the number of consistency checks performed by A, and Y the number of nodes visited by A. In total, 4050 6-tuples were captured. The experiments and analysis were executed on a SUN SPARCstation IPC, and the software was compiled SUN CLOS 4.0, developed under SPE.

5. ANALYSIS OF RESULTS

In analyzing the experimental data we first attempt to rank the nine algorithms with respect to consistency checks performed, and then with respect to the number of nodes visited. We then investigate the relationship between consistency checks and the topological parameters $W^*(d)$ and $B(d)$. Finally, we look at the run times of the algorithms.

Table 1 shows the performance of the algorithms with respect to the number of consistency checks performed. The column μ is the average number of consistency checks performed over the 450 instances of the ZEBRA problem, σ is the standard deviation, Min is the minimum number of consistency checks performed (the best case over the 450 instances), Max is the maximum number of consistency checks performed (the worst case over the 450 instances).

Therefore, we may rank the algorithms as follows: FC-CBJ < FC-BJ < BM-CBJ < FC < CBJ < BMJ < BM < BJ < BT, where "<" is interpreted as "on average performs less consistency checks than." The algorithms were compared, one against the other, on each problem instance. An entry in Table 2 shows the number of times algorithm X (row X) performed less consistency checks than algorithm Y (column Y), over the 450 problem instances. A table entry was computed as $\Sigma_{i=1}^{450}$ if $X\ (P_i) < Y\ (P_i)$ *then* 1 *else* 0 (where P_i is the ith problem, and $X\ (P_i)$ and $Y\ (P_i)$ are the number of consistency checks performed by algorithms X and Y respectively when applied to that problem).

Looking at row BM we see that there were 31 instances where BM performed fewer checks than BMJ. Over these 31 instances BMJ performed on average 28% more checks than BM (and a worst case of 90% more checks). The only distinguishing feature of these problem instances was that BJ also performed worse than BM. These were eight instances where BM performed fewer checks than BM-CBJ. Over these eight instances BM-CBJ performed on average 16% more checks than BM (and a worst case of 40% more). Again, the distinguishing feature of these eight problem instances was that CBJ also performed worse than BM. In addition, (row BMJ) there were 17 instances where BMJ performed fewer checks than BM-CBJ (on average 13% more, worst case 45% more). All of these 17

TABLE 2. How Often One Algorithm (Row) Bettered Another (Column).

	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	—

instances were relatively easy problems (a maximum of 8,170 checks for BM-CBJ), and the two algorithms never differed by more than 20%. Therefore it appears from the above analysis that whenever BJ performs worse than BM (318 instances) there is a risk that BMJ will also perform worse than BM (31 instances). Similarly, when CBJ performs worse than BM (80 instances), there is a risk that BM-CBJ will perform worse than BM (8 instances). Although BJ always performed better than BT, and CBJ always performed better than BJ, this is no guarantee that FC-CBJ will always be better than FC-BJ (or that they will be better than FC). For example, there were 12 instances where FC performed the same number of checks as FC-BJ, 10 instances where FC was as good as FC-CBJ, and 62 instances where FC-BJ was as good as FC-CBJ.

Table 3 shows the performance of the algorithms with respect to the number of nodes visited. BT and BM visit the same number of nodes in the search tree. This is expected, since BM is essentially BT modified such that it attempts to avoid redundant consistency checks. Similarly BMJ and BJ visit the same number of nodes, and BM-CBJ and CBJ visit the same number of nodes. Again, these are expected results, since BMJ is derived from BJ and BM-CBJ is derived from CBJ. Again we see evidence of the effects of forward checking. The FC hybrids consistently show a low mean number of nodes visited (and a consequent reduction in the standard deviation), consistently lower than any of the "backward-checking" algorithms. We may rank the algorithms as FC-CBJ < FC-BJ < FC < BM-CBJ = CBJ < BMJ = BJ < BM = BT, where "<" is read as "on average visits less nodes than."

TABLE 3. Nodes Visited.

Algorithm	μ	σ	Min	Max
BT	746,728	1,742,012	321	15,862,302
BJ	92,842	270,606	68	3,200,564
CBJ	11,106	31,409	64	521,643
BM	746,728	1,742,012	321	15,862,302
BMJ	92,842	270,606	68	3,200,564
BM-CBJ	11,106	31,409	64	521,643
FC	4,092	8,643	29	123,403
FC-BJ	1,877	3,338	29	30,348
FC-CBJ	1,128	1,733	29	12,247

TABLE 4. Coefficients of Correlation.

Algorithm	$B(d)$	$W^*(d)$
BT	0.200 (0.322)	0.186 (0.220)
BJ	0.124 (0.263)	0.316 (0.246)
CBJ	0.094 (0.162)	0.052 (0.076)
BM	0.165 (0.342)	0.101 (0.260)
BMJ	0.131 (0.275)	0.122 (0.224)
BM-CBJ	0.105 (0.181)	0.054 (0.079)
FC	0.107 (0.156)	0.100 (0.136)
FC-BJ	0.082 (0.084)	0.045 (0.051)
FC-CBJ	0.094 (0.081)	0.036 (0.030)

The data was analyzed to determine the effects of bandwidth and induced width on the algorithms.[16] Table 4 shows the coefficients of correlation $r_{x,y} = |S_{xy}/(S_x \cdot S_y)|$. For a sample of size 450 the $P = 1\%$ value of $r_{x,y}$ is 0.122. Therefore if $r_{x,y} \geq 0.122$ we can say with 99% confidence that there is a *linear* association between x and y. Table 4 shows $r_{x,y}$ where y is consistency checks (and in parentheses y is the logarithm of consistency checks). In the second column x is bandwidth B (d), and in the third column x is induced width $W^*(d)$.

In Table 4 there is a statistically significant coefficient of correlation between checks and bandwidth for BT, BJ, BM, and BMJ, and between checks and induced width for BT, BJ, and BMJ. These are algorithms that check backward and either chronologically backtrack or jump and step back. However, when we take the logarithm of consistency checks we find a significant coefficient of correlation with bandwidth for almost all the algorithms (the exceptions being FC-BJ and FC-CBJ). Therefore, it appears that for the majority of the algorithms the search effort (measured as consistency checks) is exponential in some function of B (d). With respect to the logarithm of induced width, CBJ, BM-CBJ, FC-BJ, and FC-CBJ show no significant values of $r_{x,y}$. Therefore, it appears that B (d) is not a good predictor of search effort for FC-BJ and FC-CBJ, and neither is $W^*(d)$ for CBJ, BM-CBJ, FC-BJ, and FC-CBJ. In fact B (d) and $W^*(d)$ were not reliable predictors of search effort for any algorithm; for example, easy problems were found at high values of B (d), and difficult problems were found at low values of B (d).

We conclude this analysis with an investigation of the run times of the algorithms. It is uncommon for run times to be reported and there are a number of reasons why this is so. First, by measuring run time we may only be measuring the ability of the programmer that implemented the given algorithms or the peculiarities of the laboratory platform. Second, it may be argued that as we move to problems where the cost of evaluating constraints is high, our measure should only be the number of consistency checks performed during the search process. However, by measuring run time we get an indication of the overheads associated with particular algorithms over the ZEBRA problem, and that is the purpose of this investigation. The algorithms were applied to the 50 problems of bandwidth, 16 and the total run time was measured (in CPU seconds) for each algorithm, along with the total number of consistency checks performed over the 50 problems. The "checking rate" for each algorithm was then estimated. Table 5 shows, for each algorithm, the average number of consistency checks performed per CPU second (the checking rate),

[16]Width was not considered, the reason being that the width of the ZEBRA is either 5 or 6. This is too small a range of x values.

TABLE 5. Run Time (CPU seconds).

Algorithm	Checks/Sec	μ	Max
BT	11,973	322	8,541
BJ	8,418	59.8	2,295
CBJ	7,346	8.6	488
BM	1,151	344	15,990
BMJ	3,806	32.9	1,370
BM-CBJ	4,592	5.5	269
FC	7,569	4.7	105
FC-BJ	7,102	2.4	39
FC-CBJ	6,503	1.6	18

an estimate of the average CPU run time μ (computed as μ from Table 1 times the checking rate), and the worst-case CPU run time Max (computed as Max from Table 1 times the checking rate).

The table entries accurately reflect the observed performance of the algorithms over the ZEBRA problem. BM was the most expensive algorithm to run. The reason for this is due to the relative simplicity in evaluating constraints within the ZEBRA compared with the costs of accessing the array element *mcl*[*i*, *k*] in *bm-label* and updating *mbl* during each call to *bm-unlabel*. In fact the most costly aspect of BM is during backtracking. When a call is made to *bm-unlabel*(*i*, *consistent*) the loop of line 6 is called, updating the array element *mbl*[*j*] for j in the range $i \leq j \leq n$. BM trades consistency checks against array accesses, and in the ZEBRA this is not an advantage. The "checking rates" of BMJ and BM-CBJ are significantly better than BM. The reason for this is due to the reduction in nodes visited by these two algorithms, with a subsequent reduction in updates of *mbl*. That is, BMJ and BM-CBJ make substantially fewer calls to *bmj-unlabel* and *bm-cbj-unlabel* respectively.

It has been argued by Ginsberg (1990) and others that when it comes to ranking the algorithms more should be made of timings and less should be made of consistency checks or nodes visited. The ranking with respect to run times is FC-CBJ < FC-BJ < FC < BM-CBJ < CBJ < BMJ <BJ < BT < BM (reading "<" as "on average solves the ZEBRA in less time than"). This ranking is in broad agreement with the previous two (checks and nodes visited). We see FC overtaking BM-CBJ (FC performs more checks, but does them nearly twice as fast as BM-CBJ), and BJ and BT overtaking BM (for the same reason). Looking at run times we might group the algorithms into three lanes: in the fast lane we would have those that (on average) gave a response in less than 10 sec (FC-CBJ, FC-BJ, FC, BM-CBJ, CBJ), in the middle lane we have those that gave a response in about a minute (BMJ and BJ), and in the crawler lane we have those that take 5 min or more (BT and BM). Therefore, to stay in the fast lane we need to use FC or CBJ, and to get the greatest speed we can combine them.

6. CONCLUSION

The process of combining tree search algorithms has been described for four new algorithms: BMJ, BM-CBJ, FC-BJ, and FC-CBJ. It seems likely that this approach may be applied to other algorithms. Immediate candidates for this process might be the nine full arc consistency hybrids in Nadel (1990) and GBJ (Dechter 1990). Therefore we have (at least) 4 choices of backward move (*bt-unlabel*, *bj-unlabel*, *cbj-unlabel*, and *gbj-unlabel*) and 12 choices of forward move (*bt-label*, *bm-label*, *fc-label*, and 9 others due to Nadel). Therefore, we should be able to synthesize (at least) 48 algorithms.

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 behavior be protected? It was also noted that backmarking requires a static order of instantiation in order to maintain the integrity of its search knowledge (arrays *mcl* and *mbl*). This suggests that BM, and the BM hybrids, cannot exploit heuristic knowledge during the search process. This may be considered as a severe limitation on the worth of these algorithms. However, this is not the case with the FC hybrids. FC-BJ and FC-CBJ can exploit heuristic knowledge. The functions *fc-label*, *fc-bj-label* and *fc-cbj-label* can be modified such that they select the current variable with the assistance of some heuristic. This suggests further experiments, similar to those in Dechter and Meiri (1989).

There is room for improvement within FC, BJ, and CBJ. These algorithms can be modified such that they detect infeasible values during the search process, and remove them once and for all. For example, in FC if the instantiation $v[i] \leftarrow k$ forward checks against $v[j]$, and this annihilates *current-domain*$[j]$, and no other variable forward checks against $v[j]$, we can remove k from *domain*$[i]$. This corresponds to detecting 2-inconsistencies. A similar approach can be taken when checking backward in BJ. In CBJ, when we jump back from $v[i]$ to $v[h]$, and $v[i]$ is in conflict only with $v[h]$, we have again discovered a 2-inconsistency (and in Prosser 1992 it is shown that we may discover k-inconsistencies). There are further enhancements possible with CBJ. When jumping back from $v[i]$, over $v[j]$, to $v[h]$ we do not need to reset *current-domain*$[j]$ if *conf-setl*$[j]$ is a subset of the past variables. When this approach is taken we realize a type (a) saving as in BM. This suggests that it is possible to incorporate a partial backmarking capability into FC-CBJ. Therefore, whenever *fc-cbj-unlabel* jumps back from $v[i]$ to $v[h]$ it might return values to *current-domain*$[j]$ only when $h \leq$ *max-list*(*conf-set*$[j]$), for all j, where $h < j \leq n$. This gives us algorithm FC-PBM-CBJ, where PBM is "partial" backmarking, and a distributed version of that algorithm has been reported by Luo, Hendry, and Buchanan (1992).

It was observed, over the 450 test cases, that the "champion" was FC-CBJ, on average performing fewer consistency checks than any other algorithm and visiting fewer nodes in the search tree. In the laboratory (SUN SPARCstation IPC, SUN CLOS 4.0, ZEBRA) this resulted in the best run times. *Caveat emptor*. 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. For example, it is easy to imagine a case where BT will outperform any algorithm based on forward checking (FC, FC-BJ, and FC-CBJ). Imagine we have a problem with n variables, where each variable has a domain of size m. Assume that the first value in the domain of each of the variables is consistent with the past variables. This would result in BT performing $\Sigma_{i=1}^{n-1} i$ consistency checks, whereas FC would perform $\Sigma_{i=1}^{n-1} m \times i$ consistency checks. Although such a problem appears overly artificial, we must taken into consideration other features. If the domains of variables are large (possibly continuous) any style of forward checking may be hopeless, and backmarking would require inordinate amounts of storage for the array *mcl*. Even if the nature of the variables' domains is not an issue, we might deny ourselves the opportunity to exploit heuristic knowledge, as noted above. Therefore, when selecting one of these algorithms for a particular application the designer should take an exploratory approach. If it is an application where FC is known to perform well, we should expect the FC-CBJ will be even better. If it is an application where backward checking is required (BT or BM), again we should incorporate CBJ.

ACKNOWLEDGMENTS

I am in debt to Ole Jakob Ottestad, my final-year student, who rediscovered BMJ, BM-CBJ, and FC-CBJ with minimal assistance, and who proposed FC-BJ. I would also like to thank my colleagues Peter Hendry, Magnus Luo, and Iain Buchanan. Finally, many thanks to Rina Dechter and Bernard Nadel for their enthusiasm and encouragement, and to my reviewers for their efforts and their advice.

REFERENCES

BITNER, J. R., and E. REINGOLD. 1975. Backtrack programming techniques. Communications of the ACM, **18**:651–656.

BURKE, P., and P. PROSSER. 1991. A distributed asynchronous system for predictive and reactive scheduling. Artificial Intelligence in Engineering, **6**(3):106–124.

DECHTER, R. 1990. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. Artificial Intelligence, **41**(3):273–312.

DECHTER, R. 1992. Constraint networks. *In* Encyclopedia of artificial intelligence, 2nd ed., vol. 1, 276–285. *Editor-in-Chief* S. C. Shapiro. Wiley-Interscience.

DECHTER, R., and I. MEIRI. 1989. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. Proceedings IJCAI-89, vol. 1, 271–277.

DECHTER, R., and J. PEARL. 1988. Network-based heuristics for constraint-satisfaction problems. Artificial Intelligence, **34**(1):1–38.

FREUDER, E. C. 1982. A sufficient condition for backtrack-free search. Journal of the ACM, **29**(1):24–32.

GASCHNIG, J. 1977. A general backtrack algorithm that eliminates most redundant tests. Proceedings IJCAI-77, vol. 1, 457.

GASCHNIG, J. 1979. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, Pittsburgh, PA.

GINSBERG, M. L., M. FRANK, M. P. HALPIN, and M. C. TORRANCE. 1990. Search lessons learned from crossword puzzles. Proceedings AAAI-90, 210–215.

GOLOMB, S. W., and L. D. BAUMERT. 1965. Backtrack programming. Journal of the ACM, **12**:516–524.

HARALICK, R. M., and G. L. ELLIOTT. 1980. Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence, **14**:263–314.

KUMAR, V. 1992. Algorithms for constraint satisfaction problems: a survey. AI Magazine, **13**(1):32–44.

LUO, Q. Y., P. G. HENDRY, and J. T. BUCHANAN. 1992. A new algorithm for dynamic distributed constraint satisfaction problems. Proceedings of the 5th Florida Artificial Intelligence Research Symposium (FLAIRS), 52–56.

MACKWORTH, A. K. 1977. Consistency in networks of relations. Artificial Intelligence, **8**:99–118.

MACKWORTH, A. K. 1992. Constraint satisfaction. *In* Encyclopedia of artificial intelligence, 2nd ed., vol. 1, 285–293. *Editor-in-Chief* S. C. Shapiro. Wiley-Interscience.

MESEGUER, P. 1989. Constraint satisfaction problems: an overview. AICOM 2(1):3–17.

MONIEN, B., and I. H. SUDBOROUGH. 1980. Bounding the bandwidth of NP-complete problems. *In* Lecture Notes in Computer Science. *Edited by* G. Goos and J. Hartmanis. Graphtheoretic Concepts in Computer Science. Proceedings of the International Workshop WG80, Bad Honnef, 279–292.

NADEL, B. A. 1989. Constraint satisfaction algorithms. Computational Intelligence, 5(4):188–224.

NUDEL, B. A. 1983. Consistent-labelling problems and their algorithms: expected complexities and theory based heuristics, Artificial Intelligence (Special Issue on Search and Heuristics), **21**(1,2):135–178. Also in Search and heuristics, North-Holland, Amsterdam, The Netherlands.

OTTESTAD, O. J. 1991. A survey of 10 CSP algorithms. Final Year Report, for the B.Sc. degree in Computer Science, Dept. Computer Science, University of Strathclyde, Scotland.

PROSSER, P. 1989. A reactive scheduling agent. Proceedings IJCAI-89, vol. 2, 1004–1009.

PROSSER, P. 1990. Distributed asynchronous scheduling, Ph.d. thesis, Dept. Computer Science, University of Strathclyde, Scotland.

PROSSER, P. 1992. Backjumping revisited. Research Report AISL-47-92, Dept. Computer Science, University of Strathclyde, Scotland.

ROSIERS, W., and M. BRUYNOOGHE. 1987. Empirical study of some constraint satisfaction algorithms. *In* Artificial intelligence. II: Methodology, systems, applications. *Edited by* P. Jorrand and V. Sgurev. Elsevier Science Publishers, North-Holland.

SMITH, B. M. 1992. How to solve the zebra problem, or path consistency the easy way. Proceedings ECAI-92, 36–37.

TSANG, E. P. K. 1992. Informal proof of completeness of CBJ. Personal communication.

VOSS, A., W. KARBACH, U. DROUVEN, and R. SHUCKEY. 1990. Operationalization of a synthetic problem. ESPRIT Basic Research Project P3178, German National Research Centre for Computer Science (GMD).

WALKER, R. L. 1960. An enumerative technique for a class of combinatorial problems. Combinatorial Analysis (Proceedings of the Symposium on Applied Mathematics, vol. X), American Mathematical Society, Providence, RI, 91–94.

ZABIH, R. 1990. Some applications of graph bandwidth to constraint satisfaction problems. Proceedings AAAI-90, vol. 1, 46–51.

APPENDIX A. PROGRAMMING CONVENTIONS

The algorithms are written in a pseudocode modeled on a combination of Pascal and Common Lisp Object System (CLOS) and are essentially an ehanced version of the language described in Appendix II of Nadel (1989).

- The assignment operator $\leftarrow$ has been used in place of the more conventional := (becomes equal to).
- All reserved words are written in uppercase (such as BEGIN, END, FOR) with the exceptions of *nil*, *true*, and *false*.
- The FOR-WHILE loop is used extensively. The form of this construct is as follows:

  FOR $v \leftarrow$ *lower* TO *upper* WHILE *condition* DO *body*

  The loop initializes the variable v to be the integer value *lower*. If $v \leq$ *upper* and *condition* is *true* then *body* is executed. On each subsequent iteration of the loop the variable v is incremented, and if (i) $v \leq$ *upper*, and (ii) *condition* = *true*, then *body* is executed again. The loop terminates when either $v >$ *upper* or *condition* is *false*. On termination of the loop, v is available and retains its most recent value. Thus, the loop terminates with v having a value one greater than after the last execution of *body*.
- It has been assumed that iteration is allowed over a list. Assume S is a finite list of discrete values:

  FOR $v \leftarrow$ EACH ELEMENT OF S WHILE *condition* DO *body*

  On first executing the above construct, the *condition* is tested, and if *condition* is *true*, v is then assigned the first element of the list S and *body* is executed. On each subsequent iteration of the loop the *condition* is tested, and if (i) *condition* is *true*, and (ii) the list has not been exhausted, then v is assigned the next element from the list S and *body* is executed again. The loop terminates when either *condition* is *false* or the list S has been exhausted. On termination of the loop, v is available, and v retains its most recent value.

- It is assumed that all parameters to a function or procedure are treated as reference variables.
- The statement *return*(x) terminates a function and delivers as a result the value of x.
- Semicolons are used to terminate successive statements. A statement is not terminated by a semicolon if it is terminated by an END.
- For the sake of brevity, type declarations of variables are assumed to be implicit. Therefore the first occurrence of a variable is taken as an implicit declaration.
- It is assumed that the language has a list processing capability, and that the language performs garbage collection. The list processing functions are described below.

list: *list* constructs and returns a list of its arguments. For example: $x \leftarrow$ *list*(1,2,3,4) assigns to the variable x the list $\langle 1\ 2\ 3\ 4\rangle$. In the functions that follow it will be assumed that the list $x = \langle 1\ 2\ 3\ 4\rangle$.

push: *push*(e,l) pushes the element e onto the list l and delivers as a result the modified list l. For example: let $y \leftarrow$ *list*(1,2,3). A call to *push*(3,y) delivers as a result the list $\langle 3\ 1\ 2\ 3\rangle$ and $y = \langle 3\ 1\ 2\ 3\rangle$.

pushnew: *pushnew*(e,l) pushes the element e onto the list l if e is not already a member of l. Therefore *pushnew*(e,l) is equivalent to

IF not member(e,l) THEN push(e,l)

For example: let $y \leftarrow$ *list*(1,2,3,). A call to *pushnew*(3,y) delivers as a result the list $\langle 1\ 2\ 3\rangle$ and $y = \langle 1\ 2\ 3\rangle$.

pop: *pop*(l) delivers as a result the first element of the list l and destructively removes that element from the list l. For example: let $y \leftarrow$ *list*(1,2,3,4). A call to *pop*(y) will deliver the result 1, and y is now $\langle 2\ 3\ 4\rangle$.

remove: *remove*(e,l) delivers as a result the list l with the first occurrence of the element e removed. For example: let $y \leftarrow$ *list*(1,2,3,2). A call to *remove*(2,y) delivers the list $\langle 1\ 3\ 2\rangle$, and $y = \langle 1\ 2\ 3\ 2\rangle$ (that is, y is not modified).

set difference: *set-difference*($l1,l2$) delivers as a result the list of elements in $l1$ which are not in $l2$. For example: let $x \leftarrow$ *list*(1,2,3,4) and $y \leftarrow$ *list*(2,3,5). A call to *set-difference*(x,y) delivers the list $\langle 1\ 4\rangle$.

union: *union*($l1,l2$) delivers as a result a new list containing everything that is an element of either of the lists $l1$ and $l2$. For example: let $x \leftarrow$ *list*(1,4,3) and $y \leftarrow$ *list*(1,3,5). A call to *union*(x,y) delivers as a result the list $\langle 1\ 4\ 3\ 5\rangle$.

max-list: *max-list*(l) delivers the largest value in a list of integers. If 1 is empty (*nil*) then *max-list(nil)* delivers-*max-int*.