Domain Independant Heuristics in Hybrid Algorithms for CSP's

by

Paul van Run

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1994

Abstract

Over the years a large number of algorithms has been discovered to solve instances of CSP problems. In a recent paper Prosser [9] proposed a new approach to these algorithms by splitting them up in groups with identical forward (Backtracking, Backjumping, Conflict-Directed Backjumping) and backward (Backtracking, Backmarking, Forward Checking) moves. By combining the forward move of an algorithm from the first group and the backward move of an algorithm from the second group he was able to develop four new hybrid algorithms: 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).

Variable reordering heuristics have been suggested by, among others, by Haralick [6] and Purdom [11, 14] to improve the standard CSP algorithms. They obtained both analytical and empiral results about the performance of these heuristics in their research.

In this thesis variable reordering heuristics are introduced into the new hybrid algorithms by Prosser and emperical results are presented about the performance of these adapted versions. Four new algorithms are derived this way: BMJ with variable reordering (BMJvar), BM-CBJ with variable reordering (BM-CBJvar), FC-BJ with variable reordering (FC-BJvar) and FC-CBJ with variable reordering (FC-CBJvar). As comparison, variable reordering is also incorporated in the standard algorithms, resulting in already known algorithms like BTvar, BMvar and FCvar, and new algorithms like BJvar and CBJvar.

Three different kinds of problems were used to obtain the emperical test results: the Zebra problem which was also used by Prosser, the N-queens problem and random problems, all with fixed domain sizes.

The empirical results indicate that variable reordering heuristics offer a significant improvement for many of these algorithms. However, they also show that for problems with fixed domain sizes the new hybrid algorithms developed by Prosser do not offer any improvements, compared to the traditional algorithms, after the incorporation of these heuristics.

Acknowledgements

I would like to thank the Verenigde Spaarbank Fund (VSB Fonds) for granting me the support of their scholarship for my studies in Canada and the Institute for Computer Research (ICR) and the Information Technology Research Centre (ITRC) for their financial support through the ICR/ITRC fellowship award in the summer and fall of 1993.

Thanks to Professor Fahiem Bacchus for taking me on as a Graduate student and for his guidance during my research and the writing of this thesis. It was a pleasure working with you.

Thanks to my roommates Derek Euale and Dexter Craig for being my first two friends in Canada. Thanks to Lisa Weatherill, for being my girl, and my best friend.

Special thanks to my parents back in the Netherlands for their financial and moral support even though it must have been very hard for them to see me leave.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A large number of problems in artificial intelligence, operations research and symbolic logic can be viewed as special cases of the general Constraint Satisfaction Problem (CSP). This problem is also known as the consistent labeling problem and examples of its use can be found in machine vision, belief maintenance, scheduling, planning, database consistency checking, temporal reasoning, graph problems, satisfiability and similar problems. The problem is known to be NP-complete.

In its general form, a CSP consists of a finite list of n variables $S = \{V_1, \ldots, V_n\}$, where each variable i has a domain $D_i = \{V_{i1}, \ldots, V_{iM_i}\}$ associated with it consisting of M_i values. Related to these variables is a set of constraints $C = \{C_1, \ldots, C_m\}$ each specified over a subset of the n variables in S. These constraints limit the possible combinations of values that the variables in that subset can take. To solve the problem we have to find an assignment of values to the variables that satisfies all the constraints simultaneously. We can also be interested in all such solutions.

The *arity* of a CSP is determined by the arity of its constraints. An n-ary constraint involves restrictions to n different variables. The arity of the CSP is equal to the maximum of the constraint arities. For instance, if constraints exist between some pairs of variables, but not between triples or larger subsets of the n variables then we have an arity of 2. CSPs of this kind are called binary. The *degree* of a CSP is determined by the size of the variable domains, being equal to the maximum of its variable domain sizes. The satisfiability problem where any variable can only take one of two values, either true or false, is an example of a CSP of degree 2.

Since it is possible to convert any n-ary CSP to a binary CSP [17] we can restrict our attention to this special case. Constraints can be represented as pairs of compatible variables with corresponding values, for example $((var_1, val_1), (var_2, val_2))$. A binary constraint can then be represented as a matrix with the values of the variables along the edges and a "1" in the corresponding column and row if the values are compatible and a "0" if they are not. The constraint matrix $[C^{ij}_{kl}]$ is a bit-matrix such that $C^{ij}_{kl} = 1$ if and only if the k-th value for variable i is consistent with the l-th value for variable j, otherwise $C^{ij}_{kl} = 0$. Since $[C^{ij}_{kl}]$ is symmetrical (the constraint graph is undirected) we could restrict ourselves to those positions where $i < j$. However, since this restriction does not give us significant space savings in implementations,[1] we choose to ignore this feature. The constraint matrix can also be used to implement the domains of the different variables: if $C^{ii}_{kk} = 1$ then value k belongs to the domain of variable i, if $C^{ii}_{kk} = 0$ then k is not an element of the domain of i.

[1]In most imperative programming languages we cannot declare arrays in which the elements have different lengths, hence it is inconvenient to take advantage of this restriction.

1.1 Generate and Test

A first approach to solving a CSP can be made by using exhaustive search, also known as the generate-and-test approach. An algorithm using that approach systematically generates all the possible assignments of values to variables and then tests each one of these to see if it satisfies all the constraints. This approach will in the worst case result in the generation of $M_1 \times M_2 \times \ldots \times M_n$ possible assignments (where M_i is the domain size of variable i) which all have to be tested. When we are trying to find all solutions for a specific CSP this approach will therefore show exponential behavior in the depth of the search tree (i.e., in the number of variables). In the case of a CSP of degree two (binary variables) the search will have a worst and average case complexity of 2^n, were n is the number of variables or the depth of the search tree. When we are searching for the first solution to a CSP of this kind, this approach will only show linear behavior when the cutoff rate p, which indicates the fixed probability that the leftmost branch of a node will succeed, is close to one [19]. The formula for the average search cost in this case is:

$$av(n) = pnC_L + q(2^n - 1)(C_L + C_R + 2C_B) - qnC_B$$

Here $q = 1 - p$, and C_R, C_L, C_B are respectively the costs of traversing the left branch, the right branch and backtracking along a branch after failure. If p is close to 1 the second term will be small and the behavior of the algorithm will be fairly linear. In that case the algorithm is likely to find a solution in the leftmost branch of the tree. For smaller values of p the second term will dominate and the complexity will grow exponentially in the depth of the tree. It is this potential for exponential behavior that makes the generate and test approach unacceptable as the general solution method for CSP's.

1.2 Standard CSP Algorithms

Standard algorithms for CSP's are traditionally divided into three classes.

Tree-search algorithms This class includes Backtracking (BT) and its refinements Backjumping (BJ) (also known as Backchecking), Conflict-Directed Backjumping (CBJ) and Backmarking (BM). Algorithms of this class attempt to generate a tree of all possible assignments of values to variables, while continuously checking if the current assignment constitutes a solution. During this search they use different methods to avoid the exhaustive generation of possible assignments or to avoid performing redundant checks.

Network Consistency or Filtering algorithms This class is comprised of arc-consistency (AC) or Waltz filtering, and path-consistency (PC) algorithms. Arc-consistency or 2-consistency algorithms (also known as constraint propagation and constraint relaxation) filter the domain of a variable in such a way that any remaining value has a compatible match in the domain of any other variable. Path-consistency or 3-consistency ensures that any subnetwork of two variables is extendable to any third variable. In general, i-consistent algorithms guarantee that any consistent instantiation of $i-1$ variables can be extended to any ith variable. Filtering algorithms can only solve problems in specific cases, in general tree-search algorithms have to be additionally used to solve the simplified problem.

Combinations This third class is comprised of algorithms that combine tree-searching and filtering. Combining these two approaches is interesting because tree-search algorithms are guaranteed to find all the solutions but suffer from thrashing (explained below) while filtering algorithms can alleviate this defect but are not guaranteed to find a solution. Examples of these algorithms are Full Lookahead (FL), Partial Lookahead (PL) and Forward Checking (FC). Algorithms of this class look forward to the domain of variables that are not yet instantiated and make sure that they are consistent with the variables that already have a value.

1.2.1 An Example: the Zebra Problem

The example problem that will be used to clarify the workings of the various algorithms is the Zebra problem. This CSP problem has 25 variables that correspond to the following:

- Five colours: Red, Blue, Yellow, Green and Ivory ($V[1]$, $V[2]$, $V[4]$, $V[5]$, $V[3]$);
- Five brands of cigarettes: Old-Gold, Parliament, Kools, Lucky and Chesterfield ($V[6]$, $V[8]$, $V[12]$, $V[13]$, $V[9]$);
- Five nationalities: Norwegian, Ukranian, English, Spanisch and Japanese ($V[7]$, $V[10]$, $V[14]$, $V[15]$, $V[11]$);
- Five pets: Zebra, Dog, Horse, Fox and Snails ($V[16]$, $V[18]$, $V[22]$, $V[23]$, $V[19]$);
- Five drinks: Coffee, Tea, Water, Milk and Orange-Juice ($V[17]$, $V[20]$, $V[24]$, $V[25]$, $V[21]$);

Each of these variables has the domain $\{1, 2, 3, 4, 5\}$ representing one of five houses they belong to. The particular numbering of the variables that is used has been chosen to simplify the construction of suitable examples in the following sections. The following constraints must be satisfied:

1. Each house has a different colour, is inhabited by a single person from a specific nationality, who smokes a unique brand of cigarettes, owns a particular pet and has a preferred drink;
2. The Englishman lives in the Red house;
3. The Spaniard owns a Dog;
4. Coffee is drunk in the Green house;
5. The Ukranian drinks Tea;
6. The Green house is to the right of the Ivory house;
7. The Old-Gold smoker owns Snails;
8. Kools are smoked in the Yellow house;
9. Milk is drunk in the middle house (house 3);
10. The Norwegian lives in the first house on the left (house 1);
11. The Chesterfield smoker lives next to the Fox owner;
12. Kools are smoked in the house next to the house with the horse;
13. The Lucky smoker drinks Orange-Juice;
14. The Japanese smokes Parliament;
15. The Norwegian lives next to the Blue house.

The constraint graph generated by this problem is shown in Figure 1.1.

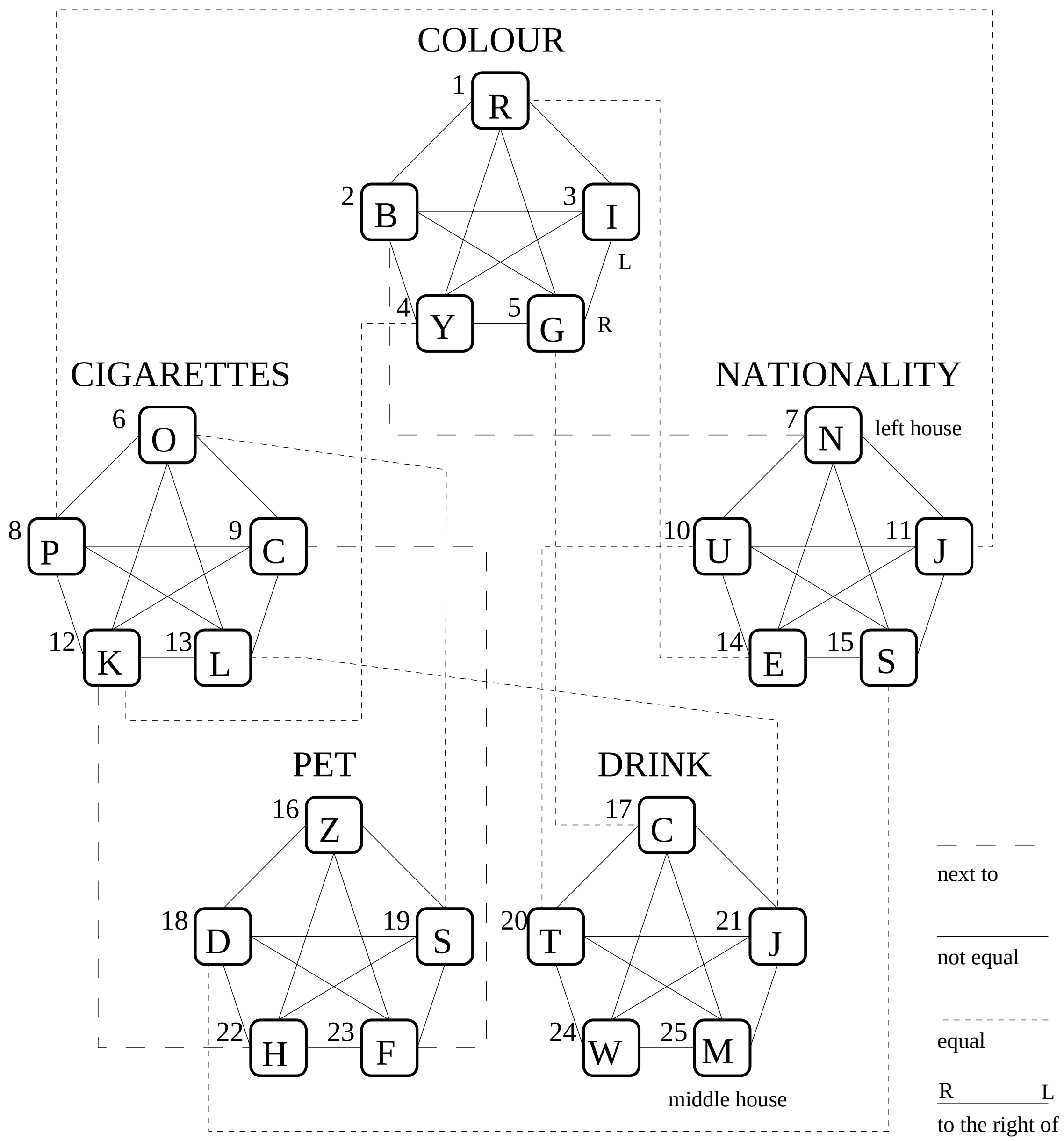


Figure 1.1: Constraint graph of the Zebra problem

1.2.2 Data Structures and definitions

The code for part of the algorithms in this work has been taken from a CSP function library by van Beek [3]. The following data structures and definitions are used in the discussion of the various algorithms that will be presented:

- *current* is the variable that is currently being instantiated, it ranges from 1 to $n+1$. After the instantiation of *current* checks out to be consistent at depth i in the search tree, current is set to $i+1$. Hence at the bottom of the tree, when a solution has been found current has the value $n+1$.
- n is the number of variables, k is the maximum domain size for these variables
- The four-dimensional array $C[n][n][k][k]$ holds the constraint matrices for all the variables. $C[i][j][k][l] = 1$ if and only if value k for variable i is compatible with value l for variable j. C also determines the domain of variable i; if $C[i][i][k][k] = 1$ then k belongs to the domain of i; if $C[i][i][k][k] = 0$ then k does not belong to this domain.
- The function *trivial*(i, j) returns $True$ if the constraint between variable i and variable j is trivial, i.e., all values of variable i are compatible with all values for variable j. This function uses information obtained from preprocessing ($O(n^2)$ algorithm) and considerably improves performance.[2]
- The array *solution*$[i]$ holds the current assignment of values to variables for $1 \leq i \leq current$. Note that this structure only holds a solution to a CSP when $current > n$. At the end of a search for all solutions to a specific CSP

[2]This function was not included in van Beek's [3] library, but was added by the author.

for example, it is unlikely that the last assignment of values to the variables constitutes a solution.

- Variable *checks* is a counter for the number of consistency checks performed. Tests that determine if a value belongs to the domain of a variable are not considered consistency checks, accessing $C[i][i][k][k]$ is therefore not counted.[3]

- Variable *found* indicates whether a solution has been found and variable *count* indicates how many solutions have been found.

- Variable *number* is used to indicate if the algorithm should search for one or for all solutions. If $number = 0$ the algorithm will look for all solutions, if $number = 1$ it will only look for the first solution.

All the algorithms that will be discussed here use the following template:

```
Function CSP-algorithm
{  If this is the first variable:
      Initialize the bookkeeping structures
   If all variables are instantiated consistently:
      Process the solution
   For all values of the variable do:
      {
      Perform domain checks
      Instantiate the variable with this value
      Perform consistency checks
      Make a recursive step to this CSP-algorithm
      Process the results of this step
      }
   Process a failure
}
```

[3]Not counting domain checks as consistency checks is in keeping with standard practice in CSP research.

A special function *consistent* checks if the instantiation of the current variable is consistent with respect to other past or future variables. (See appendix A for some programming conventions)

1.2.3 Backtracking (BT)

Chronological backtracking [20, 2] is one of the simplest algorithms for solving CSP's and serves as a basis for all of the other algorithms discussed in this thesis. In backtracking the set of all variables is instantiated incrementally, one variable at a time. When a variable is assigned a value from its respective domain a partial consistency check is performed, involving only those constraints for which all the variables are currently instantiated. If variable V_i is currently being instantiated only the constraints involving V_i and variables from $\{V_1 \ldots V_{i-1}\}$ have to be checked. Constraints that only involve variables from $\{V_1 \ldots V_{i-1}\}$ were checked previously and constraints involving variables from $\{V_{i+1} \ldots V_k\}$ cannot be checked because these variables have not yet been assigned. When any of the constraints fail the next value for V_i is tried, and when all values are exhausted V_i is unassigned and the next value for the previous variable V_{i-1} is checked.

The backtrack algorithm (BT) is given below. The *consistent* function checks if the assignment to the current variable is consistent with the previous assignments by checking all the constraints involving the variables from 1 to $current - 1$. If the constraint between the current variable and a previous one is trivial, the function skips checking it. Actually performing a check between two variables is done by accessing the C-array using these variables and their current values from the *solution*-array as indicies. Each such use of the C-array is counted as a consistency check.

```
Function consistent(C, solution, current)
   NETWORK  C;
   SOLUTION solution;
   int      current;
{  int i;
   for (i := 1; i < current; i++) {
      if (trivial(current,i)) continue;    /* skip trivial constraints */
      checks := checks + 1;             /* count the consistency check */
      /* consistency check between variable current and variable i     */
      if (C[current][i][solution[current]][solution[i]] = 0)
         return(0);}                                       /* failure */
   return(1);                                              /* success */
}
```

The main loop of the BT algorithm chronologically traverses the search tree by assigning a value from its domain to the current variable, checking its consistency and calling the function recursively for the next variable. If none of the values prove to be consistent with the past instantiations the algorithm backs up to the previous variable and tries its next value. If $current > n$ the function processes a solution as all the variables have been assigned consistent values. It then continues the search for other solutions, or it returns control to the calling program if only the first solution was requested.

```
Function BT(C, n, k, solution, current, number, found)
   NETWORK  C;
   int      n, k, current, number, *found;
   SOLUTION solution;
{  int i;
   if (current = 1)                                          /* initialize */
      *found := 0;
   if (current > n) {                                  /* found a solution */
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(1);                     /* only first solution */
      else return(0);}                                     /* all solutions */
   for (i := 1; i <= k; i++) {                          /* check all values */
      if (C[current][current][i][i] = 0)            /* if not in the domain */
         continue;                                  /* go to the next value */
      solution[current] := i;                            /* assign the value */
      if (consistent(C, solution, current)           /* check consistency */
         /* call BT recursively with current+1                              */
         if (BT(C, n, k, solution, current + 1, number, found))
            return(1);}                                           /* success */
   return(0);                                         /* failure, backtrack */
}
```

Backtracking eliminates substantial subspaces of the search space expanded by the generate-and-test algorithm by checking partial solutions to a CSP. In order to do this it needs to be able to use partial constraints. A partial constraint only involves a subset of the total set of variables. It has the property that it will always be satisfied by an assignment of values to the variables in this subset if such an assignment can lead to a solution. It *might* fail if the particular assignment cannot lead to a solution.

A constitutes a solution $\Rightarrow ((A' \subset A) \Rightarrow PC(A') = True)$

In this formula A is an assignment of values to all variables and PC is a partial constraint. A good partial constraint fails with most assignments that cannot lead to a solution. This way a single partial consistency check can prevent an

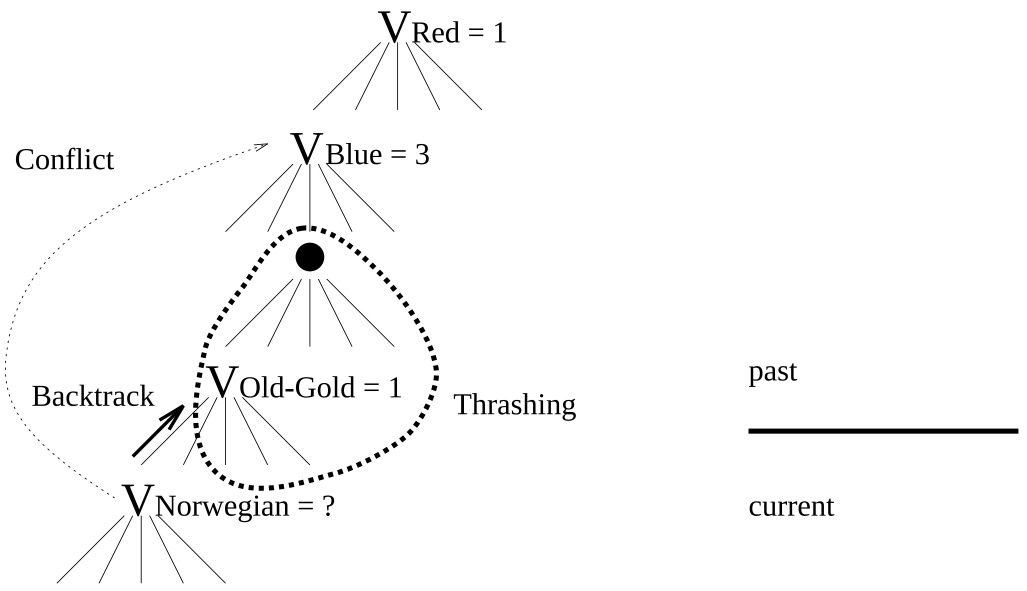


Figure 1.2: BT scenario

entire subtree, that would not lead to a solution, from being searched. However, backtracking can still be very inefficient because it may suffer from what is known as *thrashing*. This basically means that during the search redundant checks are preformed due to the incompatibility of a set of variables.

In Figure 1.2 an example of this behaviour for the Zebra problem is shown. The domain of $V_{Norwegian}$ is wiped out because according to constraint 9 he has to live in the Left house (house 1) and according to constraint 14 he also has to live next to V_{Blue}, which has value 3. The algorithm will now backtrack to $V_{Old-Gold}$, but the problem will keep occuring until V_{Blue} is assigned another value. The checks performed between $V_{Norwegian}$ and V_{Blue} are redundant.

In general, backtracking, like all the other algorithms that will be discussed here, tends to have an exponential time complexity in the number of the variables, or the depth of the tree, both in the average and worst-case.

1.2.4 Backjumping (BJ)

Backjumping is an algorithm developed by Gaschnig [5] that jumps back multiple levels, directly to the cause of a conflict to avoid thrashing. This way the number of nodes visited in the search tree can be reduced, resulting in a reduction in the number of consistency checks. If no value can be found in the domain of the current variable that is consistent with the past variables, BJ jumps back to the deepest variable in the tree that precluded a candidate value from the current domain. Its forward move is still same as in Backtracking, checking the new variable against all instantiations in the past.

```
Function consistent(C, solution, current)
   NETWORK C;
   int current;
   SOLUTION solution;
{  int i;
   for (i := 1; i < current; i++) {
      if (trivial(current,i)) continue;
      checks := checks + 1;
      if (C[current][i][solution[current]][solution[i]] = 0) {
         if (i > jump_place[current])                   /* BJ, failure */
            jump_place[current] := i;         /* BJ, update if deeper */
         return(0); }  }
   jump_place[current] := current - 1;                  /* BJ, success */
   return(1);
}
```

The *consistent* function is very similar to the one in BT except for the introduction of the *jump_place*-array which is initialized to zero for all i before the search starts. If during the consistency checking loop one of the checks fails then *jump_place*[*current*] is set to the variable causing the conflict if it is deeper in the search-tree than the present value of *jump_place*[*current*]. If all the checks succeed *jump_place* is set to *current* $- 1$. The lines marked with "/* BJ */" indicate the differences with BT.

Instead of a *True* or *False* indication the main function of BJ returns the variable number of the variable to jump back to. When a backjump occurs the *jump_place*-array has to be restored for all the variables between the spot where the algorithm will jump back to and the current variable.

```
Function BJ(C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i, jump;
   if (current = 1) {
      clear_setup(n);
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return (0);
      else return(n);}
   for (i := 1; i <= k; i++) {
      if (C[current][current][i][i] = 0) continue;
      solution[current] := i;
      if (consistent(C, solution, current)) {
         jump := BJ(C, n, k, solution, current + 1, number, found);
         if (jump <> current) return(jump);}}  /* jumpback to ''jump'' */
   jump := jump_place[current];
   for (i := jump+1; i <= current; i++){   /* restore jump_place array */
      jump_place[i] := 0;}
   return(jump);                           /* return the backjump-spot */
}
```

Figure 1.3 is basically the same as Figure 1.2 but when BJ is used the algorithm would jump back directly from $V_{Norwegian}$ to V_{Blue}, the cause of the conflict. It is important to note that only one backjump will occur and not a series of them.

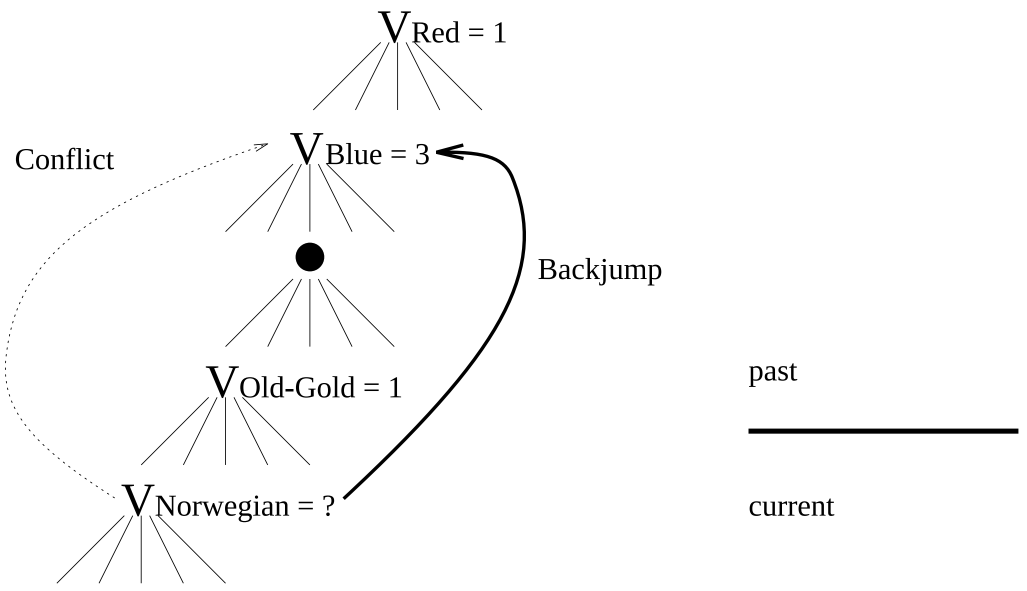


Figure 1.3: BJ scenario

If a jumpback to variable V_{Blue} occurs this means that the instantiation of V_{Blue} precluded some value from the current domain. However, the fact that V_{Blue} was instantiated in the first place means that it passed all the consistency checks with its predecessors and therefore $jump\_place[V_{Blue}] = V_{Red}$. If subsequently all the values for V_{Blue} are exhausted the algorithm would step back to V_{Red}.

1.2.5 Conflict-Directed Backjumping (CBJ)

Conflict-Directed Backjumping is an improvement of Backjumping that can handle multiple backjumps in a row. When a backjump occurs from V_i to V_h CBJ continues to jump back across conflicts that involve both V_i and V_h. This is accomplished by recording the *conflict-set* of every variable in the $conflicts[N][N]$ array. $Conflicts[i][j] = 1$ represents a conflict between V_i and V_j that pruned a value from the first variable's domain. Initially all *conflict-sets* are set to be empty.

The conflict-set holds all the past variables that failed consistency checks with the current variable. If no consistent value can be found in the domain of V_i then CBJ jumps back to the deepest variable V_h in its *conflict-set*. The *conflict-set* of V_h is then changed to be the union of its current *conflict-set* and the *conflict-set* of V_i. If a wipe-out occurs at V_h CBJ jumps back to the deepest variable in this union. The *consistent* function is very similar to BT except for the lines marked "/* CBJ */". As an extra improvement $\mathit{conflicts}[i][i]$ always holds the maximum value in the *conflict-set* of V_i. CBJ has a primitive forward move since no extra information is used while checking the new instantiation of the current variable.

```
Function consistent(C, solution, current)
   NETWORK C;
   int current;
   SOLUTION solution;
{  int i;
   for (i := 1; i < current; i++) {
      if (trivial(current,i)) continue;
      checks := checks + 1;
      if (C[current][i][solution[current]][solution[i]] = 0) {
         conflicts[current][i] := 1;          /* CBJ, conflict occurred */
         if (conflicts[current][current] < i)  /* CBJ, bigger than max */
            conflicts[current][current] := i;       /* CBJ, update max */
         return(0);}  }
  return(1);
}
```

In the CBJ function the jump is made to the deepest variable in the *conflict-set* which is stored in $\mathit{conflicts}[\mathit{current}][\mathit{current}]$.

The function $\mathit{union\_conflicts}(\mathit{jump}, \mathit{current})$ calculates the union of the *conflict-sets* of *jump* and *current*. The function $\mathit{empty\_conflicts}(\mathit{jump}, \mathit{current})$ resets these sets for all variables between *jump* and *current*.

```
Function union_conflicts(i, j)
   int i, j;
{  int m;
   for (m := 1; m < i; n++) {
      conflicts[i][m] := conflicts[i][m] or conflicts[j][m];  /* union */
      if (conflicts[i][m] and conflicts[i][i] < m)
         conflicts[i][i] := m;}     /* set conflict[i][i] to max value */
}


Function CBJ(C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i, jump, curr;
   curr := count;
   if (current = 1) {
      clear_setup(n);
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(0);
      else return(n);}
   for (i := 1; i <= k; i++) {
      if (C[current][current][i][i] = 0)
         continue;
      solution[current] := i;
      if (consistent(C, solution, current)) {
         jump := CBJ(C, n, k, solution, current + 1, number, found);
         if (jump <> current)
            return(jump);}  }
   if (curr = count)               /* we didn't come across a solution */
      jump := conflicts[current][current]      /* return jump position */
   else                                   /* we came across a solution */
      jump := current - 1;         /* return to the previous variable */
   union_conflicts(jump, current);
   empty_conflicts(jump, current);
   return(jump);
}
```

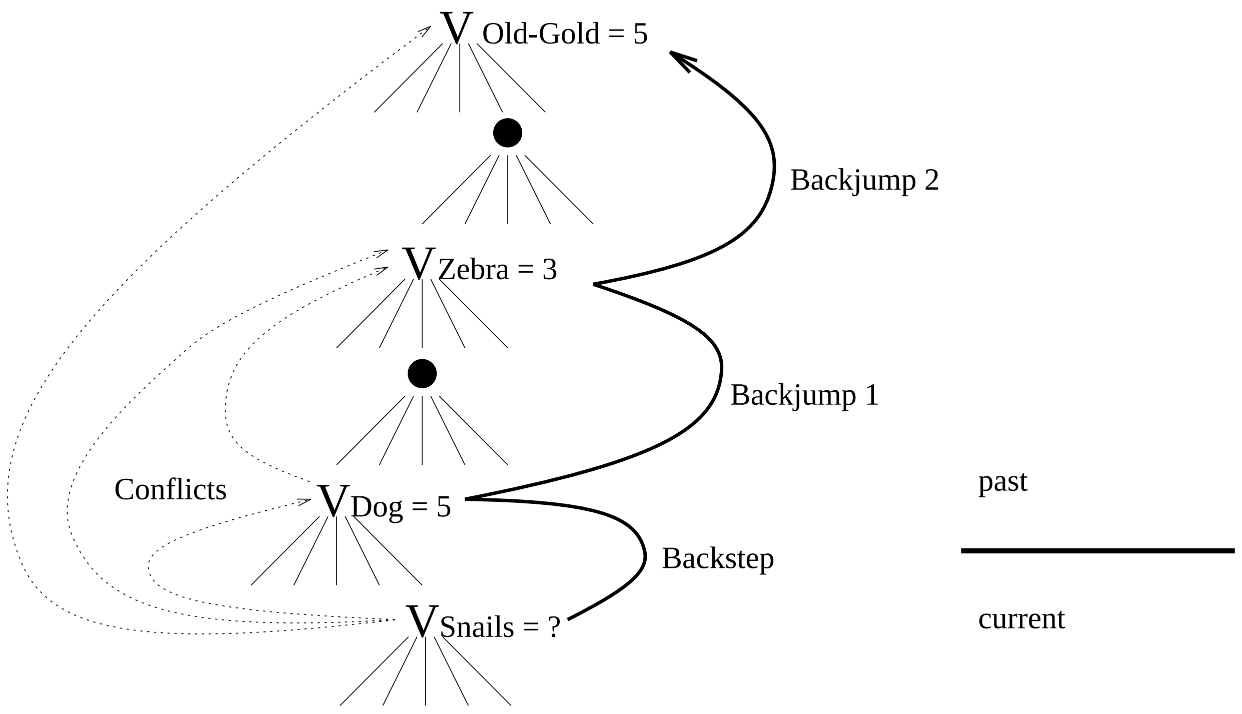


Figure 1.4: CBJ scenario

The variable *curr* is used to test whether a solution has been found on the current search path. If this is so then the other values for the current variable are tried and then the algorithm steps back to the previous variable. If the algorithm has not found a solution then the information from the *conflict*-array is used to jump back to the source of the conflict. Variable *curr* is assigned the value of *count* before the recursive step is made. If there is a difference between *curr* and *count* after the recursive call then this indicates that we have encountered a solution on the current search path.

When we apply CBJ to the scenario in figure 1.4 the algorithm would at first step back from V_{Snails} to V_{Dog}. When subsequently all the values of V_{Dog} are also exhausted it would jump back to V_{Zebra} and from there to $V_{Old-Gold}$, each time jumping back to the deepest variable in the union of the conflict sets. The backstep and the first backjump result from constraint 1, "Each house has one pet". The second backjump is made as a result of constraint 7, "The Old-Gold smoker owns Snails".

1.2.6 Backmarking (BM)

Backmarking (BM) by Gaschnig [4] is aimed at eliminating redundant constraint checks by preventing the same constraint from being tested repeatedly. This is achieved by employing two arrays *mcl* and *mbl*. The maximum checking level $mcl[i,k]$ is the deepest variable that the instantiation $V_i = k$ checked against. The minimum backup level $mbl[i]$ is the shallowest past variable that has changed its value since V_i was the current variable. BM uses a primitive kind of backward move by just stepping back to the previous variable, but its forward move is more informed than the previous algorithms.

Two situations can arise:

- Case 1, the current variable V_i is about to be re-instantiated with a value k for which a previous instantiation failed because of a conflicting variable V_j, if V_j still holds the same value the check will fail again and doesn't have to be performed;
- Case 2, the current variable V_i is about to be re-instantiated with a value k for which a previous check with variable V_j succeeded, if V_j still holds the same value the check will succeed again and doesn't have to be performed.

When $mcl[current][solution[current]] < mbl[current]$ (case 1) we know that the algorithm has not backtracked past the level were the last inconsistency occurred for this value of the current variable. It would therefore occur again and the algorithm need not consider this instantiation.

If $mcl[current][solution[current]] \geq mbl[current]$ (case 2) we know that the variables that were instantiated before $mbl[current]$ still have the same value and therefore the algorithm does not have to check them again.

```
Function consistent(C, solution, current)
   NETWORK C;
   int current;
   SOLUTION solution;
{  int i;
   if (mcl[current][solution[current]] < mbl[current])    /* BM, case 1*/
      return(0);
   for (i := mbl[current]; i < current; i++) {            /* BM, case 2*/
      mcl[current][solution[current]] := i;
      if (trivial(current,i)) continue;
         checks := checks + 1;
      if (C[current][i][solution[current]][solution[i]] = 0)
         return(0);}
   return(1);
}
```

If the domain for the current variable is exhausted $mbl[current]$ is set to $current - 1$, the previous variable. To restore the *mbl*-array the backtrack points of all the future variables have to be set to the minimum of their current value and $current - 1$, the new backtrack point.

BM has a primitive form of backward move so only the success or failure of an instantiation are returned and no backjump information.

```
Function BM(C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i;
   if (current = 1) {
      clear_setup(n, k);
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(1);
      else return(0);}
   for (i := 1; i <= k; i++) {
      if (C[current][current][i][i] = 0)
         continue;
      solution[current] := i;
      if (consistent(C, solution, current))
         if (BM(C, n, k, solution, current + 1, number, found))
            return(1);}
   mbl[current] := current - 1;
   for (i := current+1; i <= n; i++)          /* BM, restore mbl array */
      if (mbl[i] > current-1)
         mbl[i] := current-1;
   return(0);
}
```

In Figure 1.5 an example is given of the two sorts of savings that the BM algorithm can make. In case 1, at the second instantiation of $V_{Japanese}$ the values $1, 2, 3, 5$ do not have to be considered because they are guaranteed to fail since $V_{Parliament}$ still has value 4 and according to constraint 14, "The Japanese smokes Parliament", these variables have to have the same value. In case 2 the instantiation of $V_{Japanese}$ with value 4 only has to be checked against $V_{Chesterfield} \ldots V_{Ukranian}$ since the previous variables still hold the same values and constraint checks involving them succeeded before, so they will succeed again.

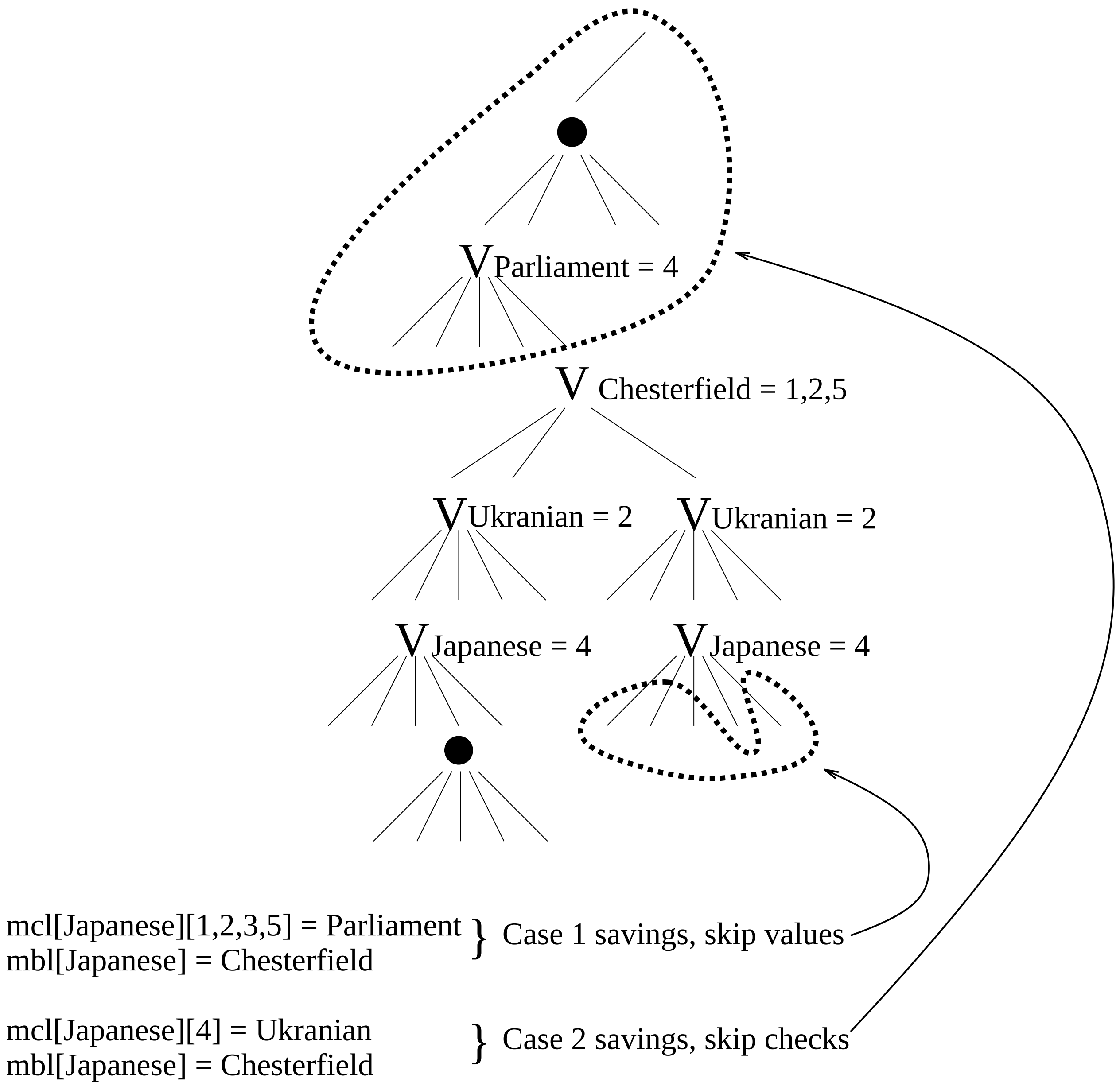

Parliament = 4
Chesterfield = 1,2,5
Ukranian = 2
Ukranian = 2
Japanese = 4
Japanese = 4
mcl[Japanese][1,2,3,5] = Parliament
mbl[Japanese] = Chesterfield
Case 1 savings, skip values
mcl[Japanese][4] = Ukranian
mbl[Japanese] = Chesterfield
Case 2 savings, skip checks

Figure 1.5: BM scenario

1.2.7 Forward Checking (FC)

Forward Checking by Haralick [6] is really a hybrid of a tree-search algorithm and a filtering algorithm. When a variable is instantiated the algorithm filters all the domains of the future variables in such a way that the remaining values in these domains are consistent with the current variable. If during this filtering process one of the domains of the future variables gets wiped out a new value for the current variable must be tried. When the FC algorithm moves forward to instantiate the next variable it does not have to perform any consistency checks because all the remaining values in the domain are guaranteed to be consistent with the past variables. FC performs more work per node, but aims at visiting less nodes in total and performing a smaller total number of checks this way.

In the implementation of this algorithm two arrays are used: $domains[N][K]$ and $checking[N][N]$. The first array keeps track of the consistency of the values in the domain of a variable. Initially all the entries are set to zero but when value k of variable V_j is pruned by the instantiation of variable V_i then $domains[j][k]$ is set to i. If variable V_i is uninstantiated during backtracking then all the values it pruned can be restored. The *checking* array keeps track of which variables pruned values from which other variables to simplify the restoration of the domains during backtracking. If $checking[i][j] = 1$ then variable i pruned a value from the domain of variable j.

There is no need for consistency checking in the *consistent* function since inconsistent values were pruned by earlier instantiations. The only case when this function returns *False* is when the domain of a future variable is wiped out during forward checking.

```
Function consistent(C, n, k, current, solution)
   NETWORK     C;
   int         n, k, current, solution;
{  int i;
   for (i := current + 1; i <= n; i++) {        /* all future variable */
      if (trivial(current,i)) continue;
      if (check_forward(C,k,current,i,solution) = 0)  /* forward check */
         return(0);}               /* a future variable got wiped out */
  return(1);
}
```

When V_j is forward checked against V_i all the values that are still in the domain of V_j are checked against the current instantiating of V_i. Values that are pruned by V_i are marked as such and if any values are pruned then the *checking* matrix is set to reflect this.

```
Function check_forward(C, k, i, j, solution)
   NETWORK C;
   int k, i, j, solution;
{  int m, old_count,delete_count;
   old_count := 0, delete_count := 0;
   for (m := 1; m <= k; n++)                               /* all values */
      if (C[j][j][m][m]  and (domains[j][m] = 0)) {/* still in domain? */
         old_count := old_count + 1;
         checks := checks + 1;
         if (C[i][j][solutiom][m] = 0) {      /* if there's a conflict */
            domains[j][m] := i;            /* prune value from domain */
            delete_count := delete_count + 1;}}
   if (delete_count)                       /* if a value was deleted  */
      checking[i][j] := 1;              /* indicate in checking array */
   /* return false if all remaining values were pruned true otherwise  */
   return(old_count - delete_count);
}
```

If V_i is uninstantiated during backtracking then all the values that V_i pruned from future variables are restored. These are the values l of V_j for which $domains[j][l] = i$. Array $checking[i][j]$ will indicate that V_i pruned values from V_j.

```
Function restore(i, n, k)
   int i, n, k;
{  int j, l;
   for (j := i + 1; j <= n; j++)                /* all future variables */
      if (checking[i][j]) {                /* i pruned some values of j */
         checking[i][j] := 0;               /* reset the checking array */
         for (l := 1; l <= k; l++)                /* for all the values */
            if (domains[j][l] = i)        /* if pruned by this variable */
               domains[j][l] := 0;}     /* restore value to domain of j */
}
```

The main function of FC is relatively simple. The domain check has been extended with *domains*[*current*][*i*]. If this array element is not equal to zero then the value *i* of the current variable has been pruned by a past variable.

```
Function FC(C, n, k, solution, current, number, found)
   NETWORK     C;
   int         n, k, current, number, *found;
   SOLUTION    solution;
{  int i;
   if (current = 1) {
      clear_setup(n, k);
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(1);
      else return(0);}
   for (i := 1; i <= k; i++) {
      if (C[current][current][i][i] = 0 or domains[current][i])
         continue;           /* skip if value not in domain or  pruned */
      solution[current] := i;
      if (consistent(C, n, k, current, solution[current]))
         if (FC(C, n, k, solution, current + 1, number, found))
            return(1);
      restore(current, n, k);}          /* restore the future domains */
  return(0);
}
```

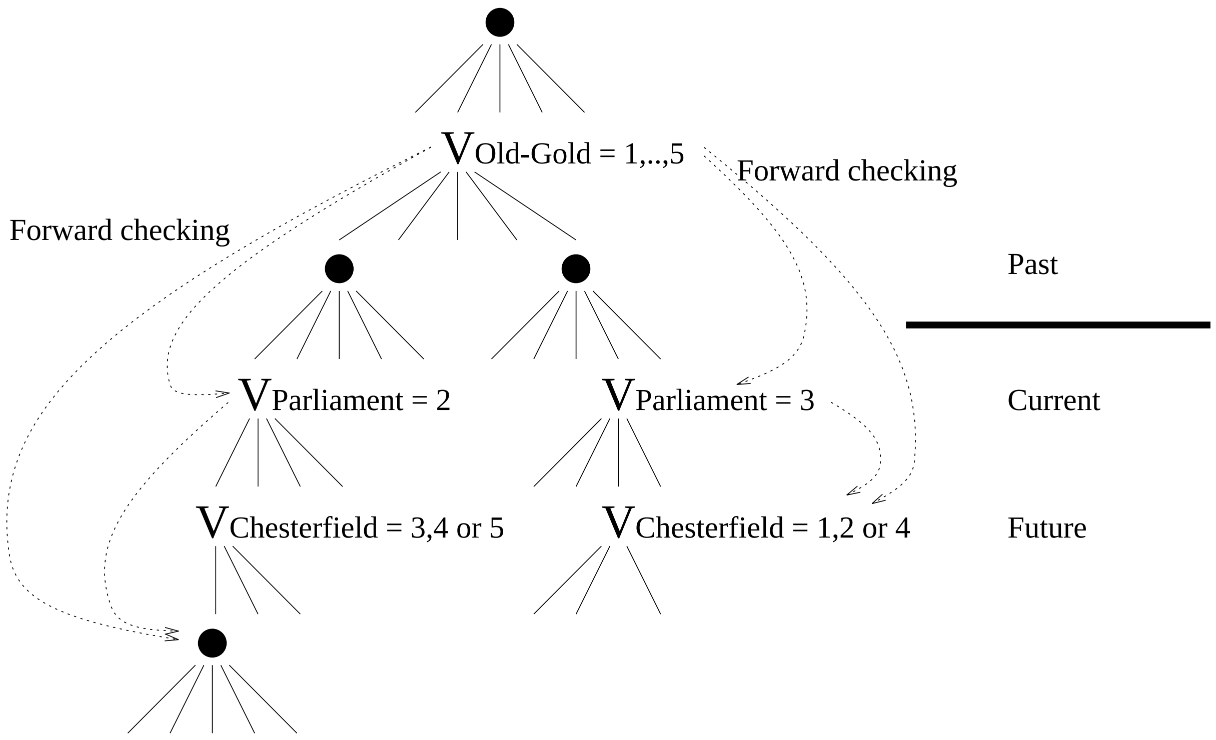


Figure 1.6: FC scenario

When the algorithm has finished investigating a certain value of a variable the function *restore* has to be called to restore the domains of the future variables.

In Figure 1.6 the assignment of 1 to $V_{Old-Gold}$ prunes this same value from the domains of $V_{Parliament}$ and $V_{Chesterfield}$ because according to constraint 1, "A different brand of cigarettes is smoked in every house". The instantiation of $V_{Parliament}$ with 2 prunes this value from $V_{Chesterfield}$ leaving only values 3, 4 and 5 as possibilities. When the algorithm backtracks the domains of the variables are restored by adding the pruned values again.

1.3 Comparing the Standard CSP Algorithms

In [6] Haralick and Elliot tested seven different CSP algorithms and found the following order (from best to worst) : word-wise Forward Checking (wFC)[4], Forward Checking (FC), Backmarking (BM), Partial Lookahead (PL), Full Lookahead (FL), Backjumping (BJ) and Backtracking (BT). The algorithms discussed in the previous sections can be ordered in a similar way as follows: FC $<$ CBJ $<$ BM $<$ BJ $<$ BT [9]. This comparison is mainly based on the number of consistency checks that the algorithms perform on several different problems. If run-time performance is considered then BM usually moves further down the ordering because of the relatively large overhead this algorithm displays.

1.4 Summary

Table 1.1 gives a summary of the algorithms discussed in this chapter.

Algorithm	forward move	backward move	next variable
BT	check against all past variables	previous variable	chronological
BJ	check against all past variables	single jump back	chronological
CBJ	check against all past variables	multiple jumps back	chronological
BM	perform only new checks	previous variable	chronological
FC	prune future variables	previous variable	chronological

Table 1.1: Summary of traditional algorithms

[4]Word-wise Forward Checking is a variant of Forward Checking which utilizes the bit-parallel capabilities of computers: several constraints are checked at once by using bit-wise *and* operations.

Chapter 2

Prosser's Hybrid Algorithms

In [9] Prosser approached the algorithms BT, BJ, BM, CBJ and FC by explicitly stating their forward and backward moves in a non-recursive fashion. In this approach BT,BJ and CBJ describe different styles of backward moves while BT, BM and FC describe different styles of forward moves. The differences between these moves is based on the amount of information that is used to make them. CBJ is more informed than BJ, and BJ is more informed than BT. The same holds for the other group, where FC is more informed than BM, and BM is more informed than BT. Simple Backtracking (BT) is considered to have the most primitive forward move, checking a new variable against all the past variables, and also the most primitive backward move, stepping back to the previous variable. By combining the backward move of an algorithm from the first group with the forward move of an algorithm from the second group Prosser developed four new algorithms: 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).

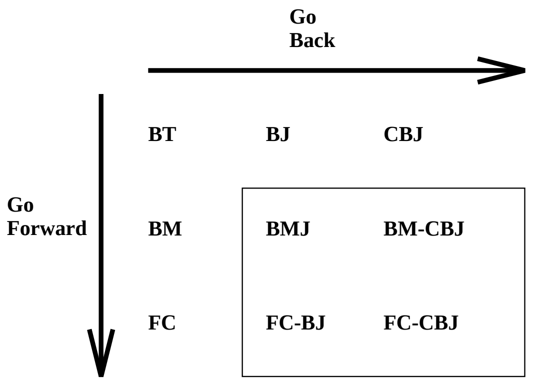


Figure 2.1: Prosser's 4 new algorithms

In the following sections recursive versions of all the algorithms in Prosser's paper [9] are given, the code of which is again partly taken from van Beek's CSP-function library [3]. The part that Prosser refers to in his paper as the *label* function can mostly be found in the *consistent* function of the recursive algorithms and the *unlabel* function is generally represented by the code segment after the recursive call.

2.1 Backmarking with Backjumping (BMJ)

Backmarking with Backjumping [9] (also known as Backmark Jumping) combines the forward move of BM with the backward move of BJ. It has the advantages of both algorithms, i.e., most of the redundant consistency checks are avoided and nodes are eliminated from the search tree by occasionally jumping back over more than one node to the source of a conflict to avoid thrashing.

The *consistent* function of BMJ is a straightforward combination of the *consistent* functions of BJ and BM. The lines labeled "/* from BM */" represent the Backmarking part of the algorithm, the lines labeled "/* from BJ */" represent the

Backjumping part.

```
Function consistent(C, solution, current)
   NETWORK C;
   int current;
   SOLUTION solution;
{  int i;
   if (mcl[current][solution[current]] < mbl[current])      /* from BM */
      return(0);
   for (i := mbl[current]; i < current; i++) {              /* from BM */
      mcl[current][solution[current]] := i;                 /* from BM */
      if (trivial(current,i)) continue;
      checks := checks + 1;
      if (C[current][i][solution[current]][solution[i]] = 0) {
         if (i > jump_place[current])                       /* from BJ */
            jump_place[current] := i;                       /* from BJ */
         return(0); } }
   jump_place[current] := current - 1;                      /* from BJ */
   return(1);
}
```

In the main function of BMJ, were the backward move of the algorithm is described, the BJ portion is present in its original form but the BM part has to be changed. In case of a backjump the maximum backup level (*mbl*) of the current variable is set to *jump*, the variable that the algorithm is jumping back to, instead of to $current - 1$, the previous variable. Both the *mbl* and the *jump_place*-array have to be restored in case of such a backjump. The *mbl* array holds information about future variables and therefore the array has to be restored from $jump + 1$ to n.

```
Function BMJ(C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i, jump;
   if (current = 1) {
      clear_setup(n, k);
      *found := 0;}
   else if (current > n) {
       process_solution(C, n, solution);
       *found := 1;
       count := count + 1;
       if (number = 1) return(0)
       else return(n); }
   for (i := 1; i <= k; i++) {
       if (C[current][current][i][i] = 0)
          continue;
       solution[current] := i;
       if (consistent(C, solution, current)) {
          jump := BMJ(C, n, k, solution, current + 1, number, found);
          if (jump <> current)
             return(jump); } }

   jump := jump_place[current];                                /* from BJ */
   mbl[current] = jump;                /* BM, jump instead of current-1 */
   for (i := jump+1 ; i <= n; i++)     /* BM, jump instead of current-1 */
      if (mbl[i] > jump)               /* BM, jump instead of current-1 *
         mbl[i] := jump;               /* BM, jump instead of current-1 */
   for (i:=jump+1;i<=current;i++)                              /* from BJ */
      jump_place[i] := 0;                                      /* from BJ */
   return(jump);
}
```

In Figure 2.2 a scenario for BMJ is given. When the algorithm tries to instantiate $V_{Norwegian}$ for the first time a wipe-out occurs because of contraints 10 and 15, "The Norwegian lives in house 1 and he lives next to the Blue house". The BJ part of the algorithm enables it to jump back to the source of the problem, V_{Blue}. When $V_{Norwegian}$ gets instantiated a second time the BM part of the algorithm prevents

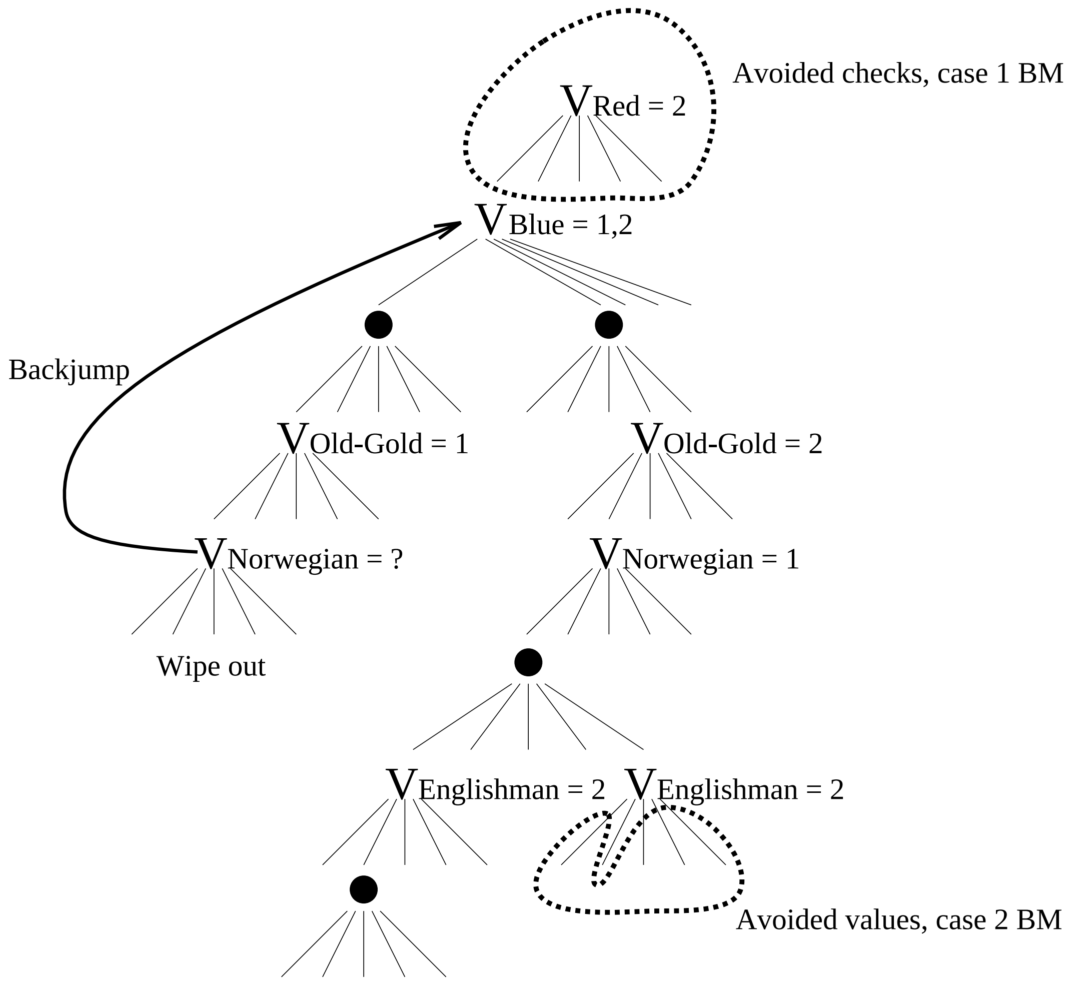


Figure 2.2: BMJ scenario

the redundant check with V_{Red} since this variable still has the same value and a check would succeed again. For an example of the second form of savings from the BM part we have to look deeper into the tree to $V_{Englishman}$. When this variable gets instantiated for the second time the algorithm can skip the values 1, 3, 4 and 5 because it found out earlier that these values are incompatible with V_{Red} according to contraint 2, "The Englishman lives in the Red house".

According to Prosser there is a scenario in which BMJ might perform worse than BM. This situation occurs when BMJ jumps from V_i, over V_h, to V_g while $mbl[h] < g$. When V_h is reinstantiated, consistency checks will be repeated between V_h and V_f for all f such that $mbl[h] \leq f < g$. Therefore the only claim that can be made is that BMJ combines *most* of the advantages of BM with BJ, and performs better in *most* cases.

2.2 Backmarking with Conflict-Directed Backjumping (BM-CBJ)

BM-CBJ is a hybrid of Backmarking and Conflict-Directed Backjumping with a less trivial construction, it tries to prevent redundant checks and has the ability of making multiple backjump.

In CBJ there was only one situation in which the *consistent* function would return *False*, only if a consistency check involving the C-array would fail. In BM-CBJ there is one other case, originating from the BM part of the algorithm, namely when $mcl[current][solution[current]] < mbl[current]$. In that case there has been a conflict in the past with the variable represented by $mcl[current][solution[current]]$. This information must be transferred to the *conflicts*-array of the CBJ part of the algorithm.

The conflict with the variable in $mcl[current][solution[current]]$ should be recorded and *conflicts*$[current][current]$ should be changed if this conflict was located deeper in the search-tree than the previous known value. The lines involved in this transfer are marked "/* Addition */" in the following code.

```
Function consistent(C, solution, current)
   NETWORK C;
   int current;
   SOLUTION solution;
{  int i;
   if (mcl[current][solution[current]] < mbl[current]) {    /* from BM */
      /* Addition for combination of BM and CBJ                     */
      conflicts[current][mcl[current][solution[current]]] := 1;
      if (conflicts[current][current] < mcl[current][solution[current]])
         conflicts[current][current] := mcl[current][solution[current]];
      return(0);}
    for (i := mbl[current]; i < current; i++) {             /* from BM */
        mcl[current][solution[current]] := i;               /* from BM */
        if (trivial(current,i)) continue;
        checks := checks + 1;
        if (C[current][i][solution[current]][solution[i]] = 0) {/* CBJ */
            conflicts[current][i] := 1;                    /* from CBJ */
            if (conflicts[current][current] < i)           /* from CBJ */
                conflicts[current][current] := i;          /* from CBJ */
            return(0); } }
    return(1);
}
```

When a backjump occurs the information for the BM part of the algorithm, array *mbl*, also has to be updated. This is done by replacing $current - 1$, the back-step point in a BM algorithm, with *jump*, the backjump point in a CBJ algorithm, just like in BMJ. The rest of the algorithm is similar to the previous ones.

```
Function BM-CBJ(C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i, jump, curr;
   curr := count;
   if (current = 1) {
      clear_setup(n, k);
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(0)
      else return(n); }
   for (i := 1; i <= k; i++) {
      if (C[current][current][i][i] = 0) continue;
      solution[current] := i;
      if (consistent(C, solution, current)) {
         jump := BM-CBJ(C, n, k, solution, current + 1, number, found);
         if (jump <> current)
            return(jump);}  }
   if (curr = count) jump := conflicts[current][current];
   else jump := current - 1;

   mbl[current] := jump;         /* from BM, jump instead of current-1 */
   union_conflicts(jump, current);                         /* from CBJ */
   for (i := jump + 1; i <= n; i++)/* from BM, jump instead of current */
      if (mbl[i] > jump) mbl[i] := jump;  /* from BM, jump for current */
   empty_conflicts(jump, current);                         /* from CBJ */
   return(jump);
}
```

The scenario that was constructed in which BMJ would perform worse than BM is also applicable to BM-CBJ. Strictly speaking BM-CBJ might be even more prone to it because of the increased number of backjumps it performs.

2.3 Forward Checking with Backjumping (FC-BJ)

FC-BJ combines the moves of FC and BJ. While the FC part of the algorithm prunes future variables of values that are not compatible with the current instantiations, the BJ part enables the algorithm to jump back over more than one variable in case of a conflict.

In FC-BJ the functions *check_forward* and *restore* are the same as for FC. The *consistent* function changes and now returns the number of the future variable whose domain was wiped out by the instantiation of the current variable. This result is used by the main function to determine to which variable the algorithm has to jump back. When all the forward checks were successful *jump_place*[*current*] is set to $current - 1$, just like in the BJ algorithm, to reflect that this variable has no conflicts with past variables.

```
Function consistent(C, n, k, current, value)
   NETWORK C;
   int n, k, current, value;
{  int i;
   for (i := current + 1; i <= n; i++) {
      if (trivial(current,i)) continue;
      if (check_forward(C, k, current, i, value) = 0)       /* from FC */
         return(i);}                         /* return wiped out variable */
  jump_place[current] := current - 1;                       /* from BJ */
  return(0);
}
```

Since there were no consistency checks performed in the *consistent* function the *jump_place*-array has to be updated in the main function of FC-BJ. This happens when *consistent* returns the variable whose domain was wiped out by the instantiation of the current variable. In this case the algorithm searches for the deepest past variable that also had a conflict with this wiped-out variable. If this variable is located deeper in the search-tree than the present value of *jump_place*[*current*] this array-element is updated. Jumping back to this variable might make certain values in the domain of the wiped-out variable available again, and this could prevent a repeat of the wipe-out that occured as a result of the previous instantiations.

The real backjump however, is also dependent on the *checking*-array. The algorithm will jump back to either a variable that also pruned values from the variable that was wiped out (case 1), or to a variable which pruned values from the current variable (case 2). The choice between these two options depends on whichever one is located deeper in the search tree.

If a backjump occurs all the values that were pruned by the variables between *jump* and *current* have to be restored. This task is performed by the *restore* function. Although the restoration does not necessarily have to be done in reverse order, it is usually done this way because this reflects the way in which the algorithm backtracks.

```
Function FC-BJ(C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i, j, jump, fail;
   if (current = 1) {
      clear_setup(n, k);
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(0)
      else return(n); }
   for (i := 1; i <= k; i++) {
      if (C[current][current][i][i] = 0 or domains[current][i]) continue;
      solution[current] := i;
      fail := consistent(C, n, k, current, solution[current]);
      if (not fail) {
         jump := FC-BJ(C, n, k, solution, current + 1, number, found);
         if (jump <> current)
            return(jump);}
      restore(current, n, k);
      if (fail)
         for (j := 1; j < current; j++)
            /* BJ adapted, case 1                                       */
            if (checking[j][fail] and jump_place[current] < j)
               jump_place[current] := j;}
   jump := jump_place[current];                               /* from BJ */
   /* BJ adapted, case 2                                                */
   for (i := 1; i <= current; i++)
      if (checking[i][current] and jump < i)  jump := i;
   for (i := current; i > jump; i--) {                        /* from BJ */
      jump_place[i] := 0;                                     /* from BJ */
      restore(i, n, k);}                                      /* from FC */
  return(jump);
}
```

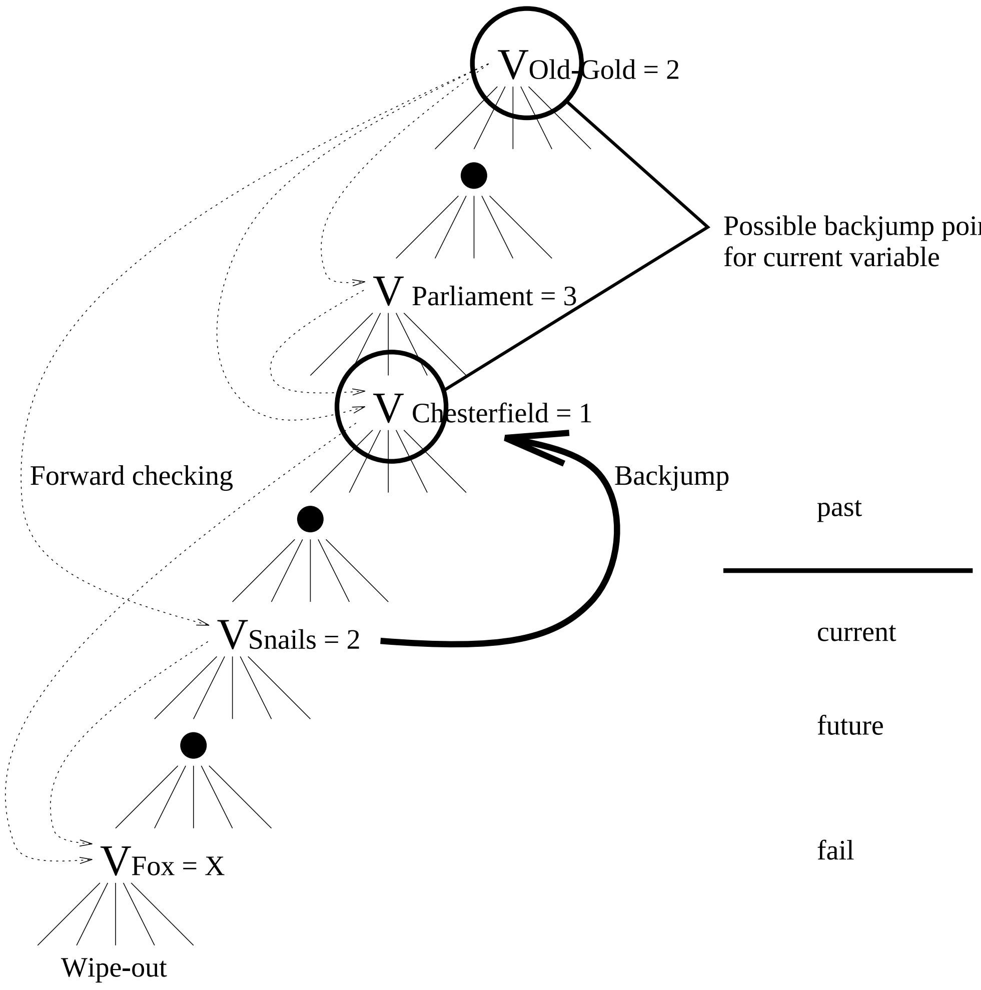


Figure 2.3: FC-BJ scenario

In Figure 2.3 an example is shown of a scenario for FC-BJ. The instantiation of $V_{Old-Gold}$ with value 2 prunes this value from the domains of $V_{Parliament}$ and $V_{Chesterfield}$ because of constraint 1 "In every house a different kind of cigarette is smoked". It also prunes the values 1, 3, 4 and 5 from the domain of V_{Snails} because constraint 7 says that "The Old-Gold smoker owns Snails". Further down the tree the instantiation of $V_{Chesterfield}$ with value 1 prunes values 1, 3, 4 and 5 from the domain of V_{Fox} because of constraint 11, "The Chesterfield smoker lives next to the

Fox owner". When the algorithm arrives at V_{Snails} it notices that the domain of V_{Fox} will be wiped out by a combination of constraints. FCBJ will now backjump to the deepest variable in the tree which pruned values from either V_{Fox} or V_{Snails}, variable $V_{Chesterfield}$ in this case.

2.4 Forward Checking with Conflict-Directed Backjumping (FC-CBJ).

FC-CBJ combines FC and CBJ and the advantages of this algorithm over FC-BJ are the same as the advantages of CBJ over BJ, i.e., the ability to jump back multiple times in a row. The algorithm forward checks all the future variables and if a wipe out occurs at a future level or at the current level the algorithm jumps back to the cause of one of these conflicts. After trying all other values for this variable it will continue to jump back using this same principle.

The functions *check_forward*, *union_conflicts* and *restore* are the same as for FC and FC-BJ. The *empty-conflicts* function resets all the conflict information for variable i. The *union_checking* function combines the information from the CBJ part of the algorithm (the *conflicts*-array) with the FC part of the algorithm (the *checking*-array), this to ensure that the correct variables will not be missed.

```
Function union_checking(i, j)
   int i, j;
{  int m;
   for (m := 1; m < i; m++)
      conflicts[i][m] := conflicts[i][m] or checking[m][j];
}
```

The *consistent* function is the same as for FC-BJ except for the removal of the BJ statements. No information is updated for the CBJ part of the algorithm so this will all have to be done in the main FC-CBJ function.

```
Function consistent(C, n, k, current, value)
   NETWORK C;
   int n, k, current, value;
{  int i;
   for (i := current + 1; i <= n; i++) {
      if (trivial(current,i)) continue;
      if (check_forward(C, k, current, i, value) = 0)
         return(i);}
  return(0);
}
```

Again, like in BJ, the algorithm has to be determined whether to jump back to the variable that helped wiping out the future variable, or to jump back to a variable that pruned a value from the current domain. Ultimately the algorithm will backjump to the deepest one of these two variables.

```
Function FC-CBJ(C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i, jump, fail, curr;
   curr := count;
   if (current = 1) {
      clear_setup(n, k);
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(0)
      else return(n); }
   for (i := 1; i <= k; i++) {
      if (C[current][current][i][i] = 0 or domains[current][i]) continue;
      solution[current] := i;
      fail := consistent(C, n, k, current, solution[current]);
      if (not fail) {
         jump := FC-CBJ(C, n, k, solution, current + 1, number, found);
         if (jump <> current) return(jump);}
      if (fail)
         union_checking(current, fail);
      restore(current, n, k);}                               /* from FC */

   if (curr =  count) {               /* didn't come across a solution */
      jump := 0;       /* set jump to the deepest variable i in either */
      for (i := 1; i < current; i++)
         if (conflicts[current][i]) jump:=i;/* conflicts[current][i] or*/
      for (i := jump + 1; i < current; i++)
         if (checking[i][current]) jump := i;}/* checking[i][current]  */
   else jump := current - 1;                 /* came across a solution */
   /* update conflict[current][] with checking[][current]            */
   union_checking(current, current);
   union_conflicts(jump, current);                           /* from CBJ */
   for (i := current; i > jump; i--) {
      empty_conflicts(i);                                   /* from CBJ */
      restore(i, n, k);}                                     /* from FC */
   restore(jump, n, k);                                      /* from FC */
   return(jump);
}
```

2.5 Summary

We can include BMJ, BM-CBJ, FC-BJ and FC-CBJ in the summary of the algorithms and their properties as shown in table 2.1:

Algorithm	forward move	backward move	next variable
BT	check against all past variables	previous variable	chronological
BJ	check against all past variables	single jump back	chronological
CBJ	check against all past variables	multiple jumps back	chronological
BM	perform only new checks	previous variable	chronological
FC	prune future variables	previous variable	chronological
BMJ	perform only new checks	single jump back	chronological
BM-CBJ	perform only new checks	multiple jumps back	chronological
FC-BJ	prune future variables	single jump back	chronological
FC-CBJ	prune future variables	multiple jumps back	chronological

Table 2.1: Summary of traditional and Prosser's algorithms

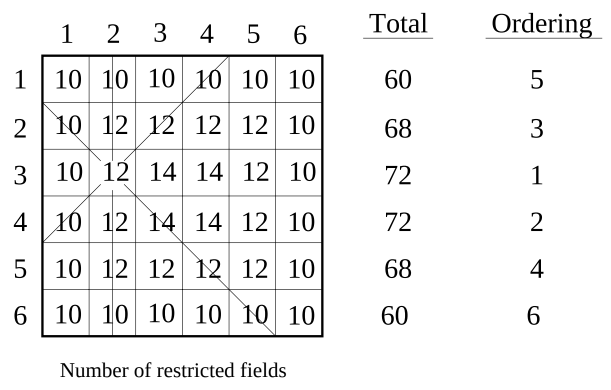
Chapter 3

Variable Reordering Heuristics

The algorithms in the previous chapter can all be improved by incorporating some form of heuristics in them. A heuristic can be seen as a general rule of thumb, a guideline to direct the search that generally improves its efficiency, but offers no guarantee that search will proceed directly to a solution. Usually a heuristic is considered to be good if it is general and cheap to use, produces solutions and prunes a large number of incorrect lines of attack [18]. There are three kind of heuristics that are often associated with CSP algorithms:

- Variable reordering: Changing the order in which the variables are instantiated.
- Value reordering: Changing the order in which the domain values of a variable are used for its instantiation.
- Constraint reordering: Changing the order in which the constraint are checked.

Combinations of these heuristics can also occur but the focus of this thesis will be on variable reordering (also known as search rearrangement) heuristics. This



	1	2	3	4	5	6	Total	Ordering
1	10	10	10	10	10	10	60	5
2	10	12	12	12	12	10	68	3
3	10	12	14	14	12	10	72	1
4	10	12	14	14	12	10	72	2
5	10	12	12	12	12	10	68	4
6	10	10	10	10	10	10	60	6

Figure 3.1: Global ordering in the n-Queens problem

heuristic involves the order in which the variables are instantiated. Instead of doing this randomly the sequence of instantiations can be ordered. This can either be done globally, before the search starts, or locally, at every node.

A global ordering orders the variables before the search commences, e.g., by selecting as variable v_k the variable which leads to the least expected number of nodes at level k in the search tree. Ties are broken by looking "deeper" into the tree and chosing the variable that allows less nodes on level $k+1$, and so on. In the n-Queens problem[1] for instance this would lead to an ordering from the middle rows outward, since a Queen in the middle row restricts the search more than one on the top or bottom of the board. In Figure 3.1 an example of a global ordering for the 6-Queens problem is given. The numbers indicate how many squares in other rows would be made unavailable if a Queen would be placed in that particular square. The columns to the right of the board indicate the totals per row and a possible ordering resulting from those totals.

The order of the variables can also be determined dynamically at every node in the tree, and vary from branch to branch, this is called a local ordering. The first

[1]For a definition of this problem see Section 4.2.2.

reference to this rule comes from Warnsdorff who applied it in 1823 to generate knight's tours. He suggested that a jump should be made to a square from which the tour had fewest possible continuations. Bitner and Reingold [2] introduced it in their backtrack algorithm by using nodes of low degree earlier in the search tree than nodes of high degree. A lot of research on local variable reordering or search rearrangement has been performed by Purdom and Brown[11, 12, 13, 14, 15].

The basic heuristic used by Bitner and Reingold examines the set of unassigned variables and instantiates the one with the fewest remaining values. This idea can be generalized to selecting a set of k unassigned variables for some predetermined number k and instantiating the variable that is the root of the smallest k-level subtree. Simple Backtracking can then be considered to be the case $k = 0$, and Bitner and Reingold's algorithm becomes $k = 1$ variable reordering Backtracking. While more consistency checks are performed per node to find the best next variable, an algorithm with variable reordering aims at visiting less nodes and thus improving performance.

3.1 Incorporating the Heuristic

To incorporate variable reordering heuristics into the existing algorithms an extra level of indirection has to be introduced. Instead of traversing the *variables* chronologically we will now traverse the *order of instantiations* of those variables. Instead of having variable i as the current variable we will be dealing with instantiation i, which can be any variable between 1 an N.

To implement this indirection an array *ins* of size N is introduced. The invariant belonging to this structure is:

$\{\forall i, j : 1 \leq i, j \leq current : ins[i] \in \{1 \ldots N\} \wedge ins[i] = ins[j] \Rightarrow i = j\}$.

During the search *ins* is a permutation of a subset of the set of variables, and when a solution is found *ins* is a permutation of the set of variables. When a variable i is selected for instantiation, $ins[current]$ is assigned value i and $solution[ins[current]]$ is assigned the first domain value of variable i. For example, if first variable 4 gets instantiated, secondly variable 7 and thirdly variable 1, then *ins* we would have: $ins[1] = 4$, $ins[2] = 7$, $ins[3] = 1$, and $ins[4] \ldots ins[N]$ would be undefined.

During the forward move of an algorithm a new variable will have to be selected that will be instantiated next. The function *next* is introduced for this purpose and it basically represents a 1-level search rearrangement heuristic. Forward checking algorithms utilize a slightly different version of this function called *nextFC* which uses the specific advantages of FC to reduce the number of checks needed to find the next variable.

It is important to note that not all bookkeeping information that is stored by the various algorithms will have to be accessed through the *ins* structure. Only information that is related to the *forward move* of an algorithm needs this extra level of indirection since the order in which future variables are accessed might be different from a order in which they were previously used.

Information needed for the backward move of an algorithm can be stored at a specific *node* in the search tree. When the algorithm backtracks this information is lost. The information needed for the forward move of an algorithm however, needs to be stored at a specific *variable*, regardless of the node in the search tree where this variable gets instantiated.

The information that is contained in the bookkeeping of the different algorithms should always pertain to *nodes* and not to *variables*. In FCvar for example, it is

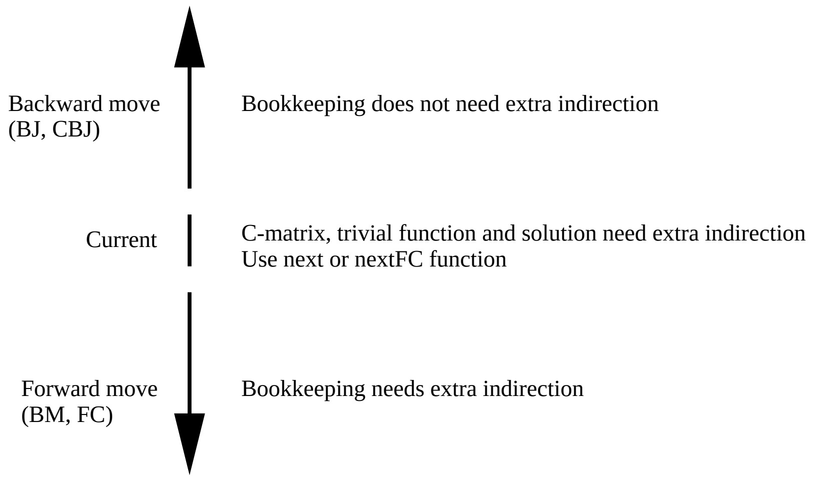

Backward move
(BJ, CBJ)
Bookkeeping does not need extra indirection
Current
C-matrix, trivial function and solution need extra indirection
Use next or nextFC function
Forward move
(BM, FC)
Bookkeeping needs extra indirection

Figure 3.2: Use of extra indirection

important to know that node 8 pruned some values from the domain of V_{Dog} so that these values can be restored if the algorithm backtracks across this node. It is not of importance to know exactly which variable did the pruning.

3.2 Standard Algorithms with Variable Reordering

Most research in this area focuses on the use of variable reordering for Backtracking and Forward Checking. Backtracking is frequently used because it is a standard algorithm for which complexity formulas can be derived. This enables us to compare the behavior of simple Backtracking with Backtracking that uses a certain heuristics analytically. Forward Checking has been the object of study, because it is one of the best algorithm known and its rather simple structure lends itself well for the incorporation of heuristics.

Introducing variable reordering into algorithms with a standard forward move such as BT, BJ and CBJ is fairly straightforward because the heuristic does not interfere with their backtrack or backjump structures. The *jump_place*-array in BJ and the *conflicts*-array in CBJ only hold information about the past for the previous and the current variables. As soon as a backjump is made all information regarding uninstantiated variables is lost. In BM and FC, the algorithms with a more informed forward move, variable reordering does interfere more directly with the stored information about the future. When one of these algorithms backtracks up one branch of a search tree and then moves forward again down another branch the ordering of the variables in the branch is likely to have changed. Therefore we have to be more cautious that the information about the future is not disturbed when chosing another ordering. Special care has to be given to the boundaries of the different loops that are used in the algorithms to restore part of the bookkeeping information when a backtrack occurs. We have to make sure that only instantiated variables get accessed through the *ins* structure, uninstantiated variables have to be dealt with directly.

In the following discussion of the different algorithms with variable reordering heuristics the emphasis will be on the changes that have to be made on those algorithms as compared to their originals. These changes are both indicated in the code segments and explaned in the accompanying text. Auxiliary function that do not need any changes are omitted and the reader is referred to the previous chapter to find their definitions.

The notion of the transformation rules given in the previous section should be sufficient to follow the process of incorporating variable reordering heuristics in the given algorithms. Especially in the more complex algorithms like BM-CBJ and FC-CBJ intuitive reasoning about how to include these heuristics gets rather complicated. Strictly following the rules however, leads to correct new algorithms.

3.2.1 Backtracking with Variable Reordering (BTvar)

The *consistent* function changes in two places. Firstly, the *trivial* function now has to access the variables through the *ins*-array. It is no longer important if variable *current* has a trivial constraint with variable i, but if the variable that was selected as number *current* in the instantiation order has a trivial constraint with variable i in that same order. Secondly, the constraint checks performed by accessing the C-array now have to use the same indirection: $ins[current]$ has to be checked against $ins[i]$. The values for these two instantiations are kept in $solution[ins[current]]$ and $solution[ins[i]]$, instead of in $solution[current]$ and $solution[i]$. Since Backtracking checks the consistency of the current variable with all the past variables we have to use as precondition for this function that ins is defined for $1 \leq i \leq current$.

```
Function consistent(C, solution, current)
   NETWORK C;
   int current;
   SOLUTION solution;
{  int i;
   for (i := 1; i < current; i++) {
      if (trivial(ins[current],ins[i])) continue; /* extra indirection */
      checks := checks + 1;
      if (C[ins[current]][ins[i]]                 /* extra indirection */
           [solution[ins[current]]][solution[ins[i]]] = 0)
         return(0);
   }
   return(1);
}
```

The function *next* is used to select the variable that will be instantiated next, this is done by counting the number of values left in the domains of the uninstantiated variables and returning the variable with the least number of values left. Ties are broken in favor of the chronological order, if variable i and j both have the same domain size then i will be selected if $i < j$. To determine if a variable is still uninstantiated the function *in_ins* is used, this function checks if a certain variable appears in the list of instantiated variables. The search for the lowest number of values is terminated as soon as a variable with zero remaining values has been found or when all possible values have been counted. Counting the values for a specific variable can be terminated as soon as its number is greater than the minimum found thus far. Special flags could be used to indicate the fact that a variable with one value or a variable with zero values left has been sighted, but in order to preserve as much of the original algorithms as possible this was not implemented. Instead *next* will just return that specific variable and leave it to the search routine to instantiate it with its only value or detect the dead-end.

```
Function next (C,ins,solution,var,nvars,k)
   NETWORK C;
   int ins[N];
   SOLUTION solution;
   int var,nvars,k;
{  int count,i,j,l,max,nt,failed;
   min := k + 1;                                   /* initialize min */
   /* count values for all variables, break off if wipe-out found      */
   for (i:=1; (i <= nvars) and (min <> 0);i++) {
      if (not in_ins(i,ins,var)) {             /* not yet instantiated */
         count := 0;
         /* check all their values, break off if count>min             */
         for (j:=1;(j<=k) and (count<min);j++) {
            if (C[i][i][j][j] = 0) failed := true;     /* in the domain */
            else failed := false;
            /* check against instantiated variables                    */
            for (l:=1; l <= var and not failed;l++) {
               checks := checks + 1;          /* count this as a check */
               if (not C[i][ins[l]][j][solution[ins[l]]])
                  failed := true;                  /* not consistent */
               else failed := false;                   /* consistent */
            if (not failed)
               count := count + 1;} /* another value left for this var */
         if (count < min) {   /* is the count smaller than the minimum */
            min := count;                          /* record the new min */
            nt := i; } } }        /* set nt to number of this variable */
   return(nt);              /* return variable with fewest values left */
}
```

The main BT function will only have a few minor changes. In the initialization the first variable that will be instantiated has to be selected. As in all the algorithms that will be discussed here this will turn out to be the variable with the smallest initial domain. The domain check and the value assignment have to be made on $ins[current]$ instead of on *current* and to determine the next variable the function *next* has to be called.

```
Function BTvar(C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i;
   if (current = 1) {
      ins[1] := next(C,ins,solution,0,n,k);   /* select first variable */
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(1);
      else return(0);}
   for (i := 1; i <= k; i++) {
      /* check domain of ins[current] instead of current              */
      if (C[ins[current]][ins[current]][i][i] = 0) continue;
      solution[ins[current]] := i;     /* assign value to ins[current] */
      if (consistent(C, solution, current)) {
         /* select the next variable                                  */
         ins[current + 1] := next(C,ins,solution,current,n,k);
         if (BTvar(C, n, k, solution, current + 1, number, found))
            return(1); } }
   return(0);
}
```

In Figure 3.3 an example is shown of BTvar. The first two variables that get instantiated are $V_{Norwegian}$ and V_{Milk} because these variables only have one value in their domain due to constraints 9 and 10: "Milk is drunk in the middle house" and "The Norwegian lives in the first house on the left". The ordering of variables with equal domain sizes is based on the underlying chronological ordering and therefore $V_{Norwegian}$ is used before V_{Milk}. V_{Blue} is instantiated next as it only has one possible value left due to the instantiation of $V_{Norwegian}$, since constraint 15 states that: "The Norwegian lives next to the Blue house". V_{Red}, V_{Ivory} and V_{Yellow} are the logical next steps on the leftmost path of the shown search tree. When the algorithm backtracks to V_{Ivory}, and assigns value 4 to the variable, V_{Green} is chosen as next

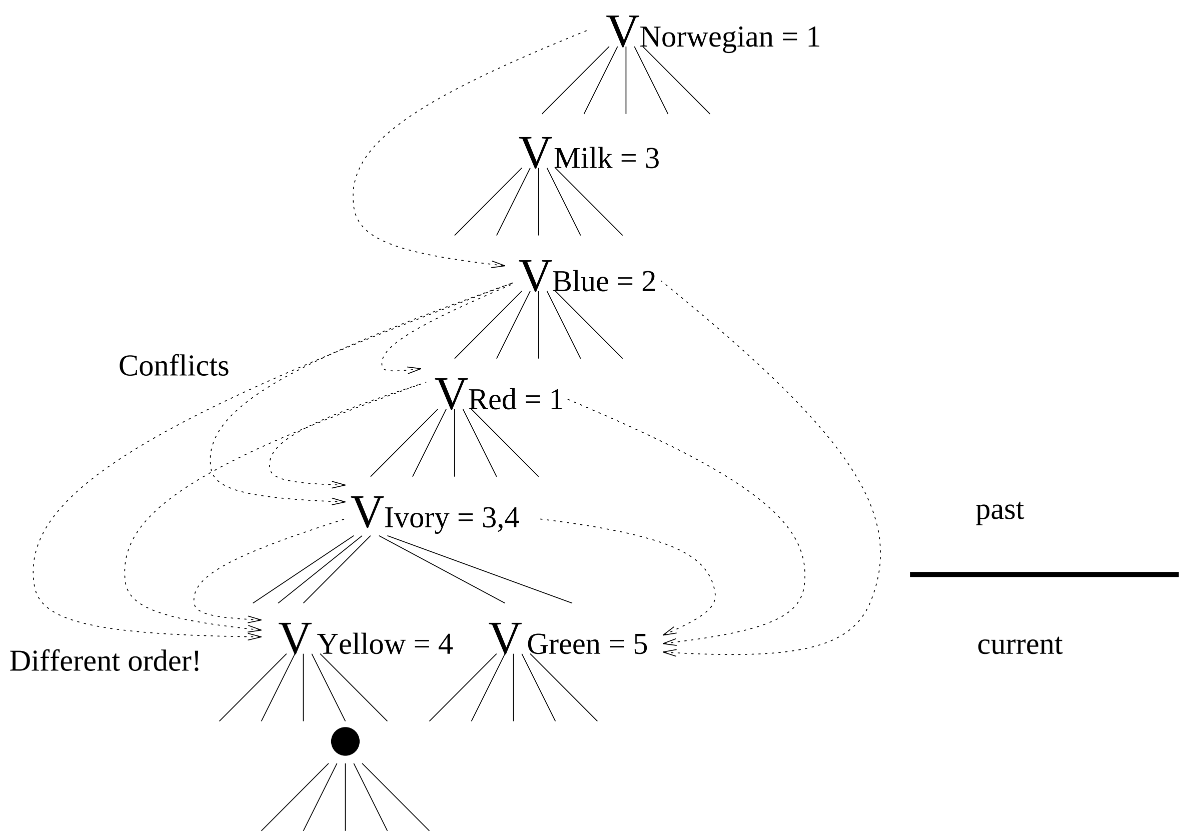


Figure 3.3: BTvar scenario

variable instead of V_{Yellow} because it only has one possible value left as a result of constraint 6: “The Green house is to the right of the Ivory house”. The different ordering does not pose any problems since BTvar uses only a primitive kind of forward move.

3.2.2 Backjumping with Variable Reordering (BJvar)

The *consistent* function in BJvar is not very different from the one in simple BJ. Again, only the *trivial* function and the constraint matrix C get an extra level of indirection. The *jump_place*-array does not need this indirection because it only has information about the past and this information will not change as a result of the variable reordering. The meaning of this array does change however, *jump_place*[*current*] is no longer a reference to the deepest variable with which the *current variable* had a conflict, but to the deepest instantiation with which the *current instantiation* had a conflict.

```
Function consistent(C, solution, current)
   NETWORK C;
   int current;
   SOLUTION solution;
{  int i;
   for (i := 1; i < current; i++) {
      if (trivial(ins[current],ins[i])) continue; /* extra indirection */
         checks := checks + 1;
      if (C[ins[current]][ins[i]]                 /* extra indirection */
           [solution[ins[current]]][solution[ins[i]]] = 0) {
         if (i > jump_place[current])
            jump_place[current] := i;
      return(0); } }
   jump_place[current] := current - 1;
   return(1);
}
```

The code of BJvar is very similar to BJ except for the same changes that were discussed for BTvar, the initialization of the first variable, the extra indirection for the constraint matrix and a call to the *next* function.

```
Function BJvar(C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i, jump;
   if (current = 1) {
      clear_setup(n);
      ins[1] := next(C,ins,solution,0,n,k);   /* select first variable */
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if(number = 1) return(0);
      else return(n);}
   for (i := 1; i <= k; i++) {
      if (C[ins[current]][ins[current]][i][i]=0)  /* extra indirection */
         continue;
      solution[ins[current]] := i;                /* extra indirection */
      if (consistent(C, solution, current)) {
         /* select the next variable                                   */
         ins[current + 1] := next(C,ins,solution,current,n,k);
         jump := BJvar(C, n, k, solution, current + 1, number, found);
         if (jump <> current)
            return(jump); } }
   jump := jump_place[current];
   for (i:=jump+1; i <= current; i++){
      jump_place[i] := 0;}
   return(jump);
}
```

Unfortunately BJvar will not offer any improvements over BTvar. BJ was an improvement of BT because it had the ability to jump back to the deepest node i that was checked against when a domain wipe-out occured at the current node j,

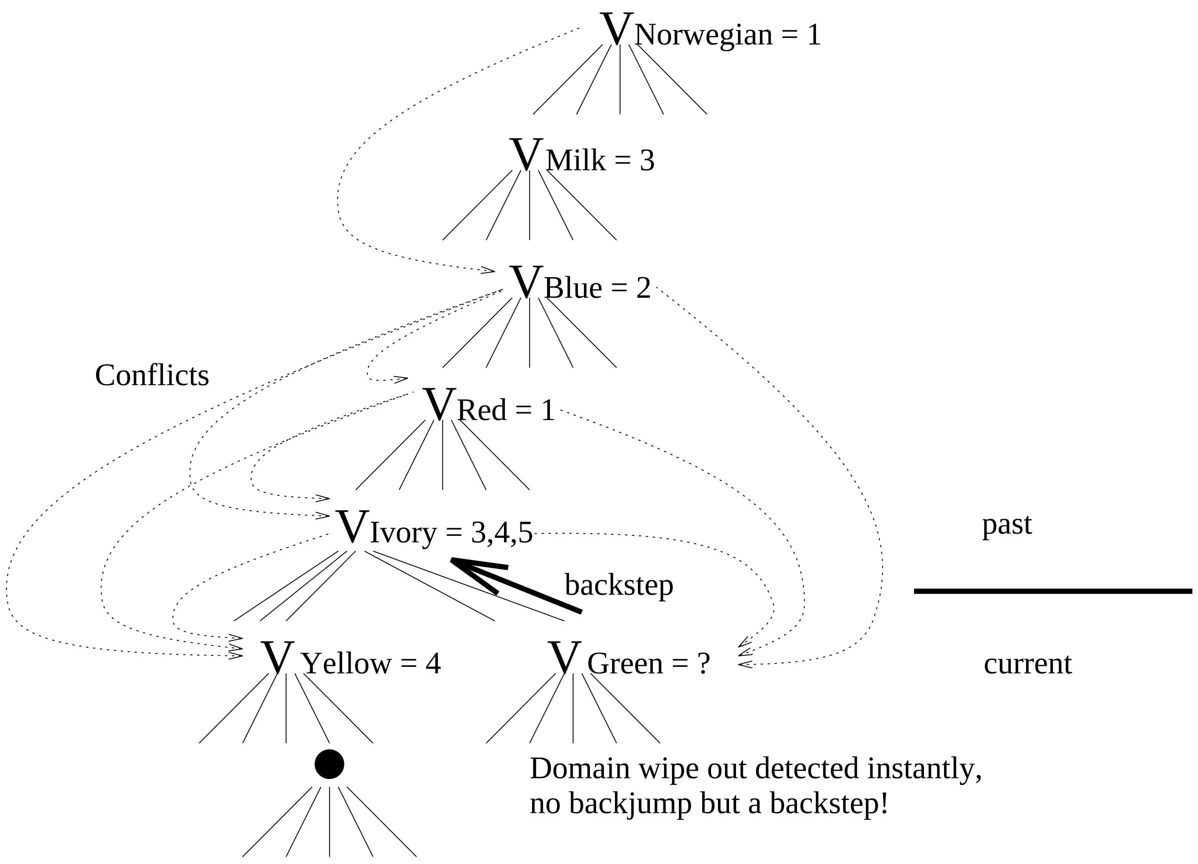


Figure 3.4: BJvar scenario

thus jumping back to the cause of the conflict. This prevented the algorithm from searching the subtree between i and j. As long as i is instantiated with its current value this subtree was not going to give a solution because the search would fail again as soon as the domain wipe-out at j occured again. In BJvar such backjumps will not occur, because if the current instantiation of i wipes out the last remaining values in the domain of future variable j, the heuristic will choose this variable as the one to be instantiated next. The search will fail and the backjump will only consist of a *backstep* to the previous node and therefore not offer any improvements over BTvar.

An example of this behaviour is shown in Figure 3.4. When V_{Ivory} is instantiated with value 5 the domain of V_{Green} is wiped out. The *next* function notices this and returns this variable as the one to be instantiated next. The search will fail and the algorithm will step back to V_{Ivory}. The resulting graph is identical to the one in Figure 3.3 as predicted.

3.2.3 CBJ with Variable Reordering (CBJvar)

Since CBJ also only keeps information about the past there is no interference with the variable reordering heuristic. In fact *union_conflicts* and *empty_conflicts* are exactly the same as in the original algorithm and the *consistent* function only needs minor changes.

```
Function consistent(C, solution, current)
   NETWORK C;
   int current;
   SOLUTION solution;
{  int i;
   for (i := 1; i < current; i++) {
      if (trivial(ins[current],ins[i])) continue; /* extra indirection */
      checks := checks + 1;
      if (C[ins[current]][ins[i]]                 /* extra indirection */
           [solution[ins[current]]][solution[ins[i]]] = 0) {
         conflicts[current][i] := 1;
         if (conflicts[current][current] < i)
            conflicts[current][current] := i;
         return(0); } }
   return(1);
}
```

The main function for CBJvar is almost identical to the one for CBJ and only in a few places do we need an extra level of indirection.

```
Function CBJvar(C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i, jump, curr;
   curr := count;
   if (current = 1) {
      clear_setup(n);
      ins[1] := next(C,ins,solution,0,n,k);   /* select first variable */
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(0);
      else return(n);}
   for (i := 1; i <= k; i++) {
      if (C[ins[current]][ins[current]][i][i] = 0)/* extra indirection */
         continue;
      solution[ins[current]] := i;
      if (consistent(C, solution, current)) {
         /* selct next variable                                     */
         ins[current + 1] := next(C,ins,solution,current,n,k);
         jump := CBJvar(C, n, k, solution, current + 1, number, found);
         if (jump <> current){
            return(jump);} } }
   if (curr = count ) jump := conflicts[current][current];
   else jump := current - 1;
   union_conflicts(jump, current);
   empty_conflicts(jump, current);
   return(jump);
}
```

In contrast to BJvar, variable reordering heuristics *can* be used with hybrids involving CBJ because CBJ records the *set* of past nodes that failed consistency test with the current node. When a domain wipe-out occurs at the current node the initial scenario will be the same as described for BJvar, the node that pruned the last remaining values from the domain of the current node will be the previous

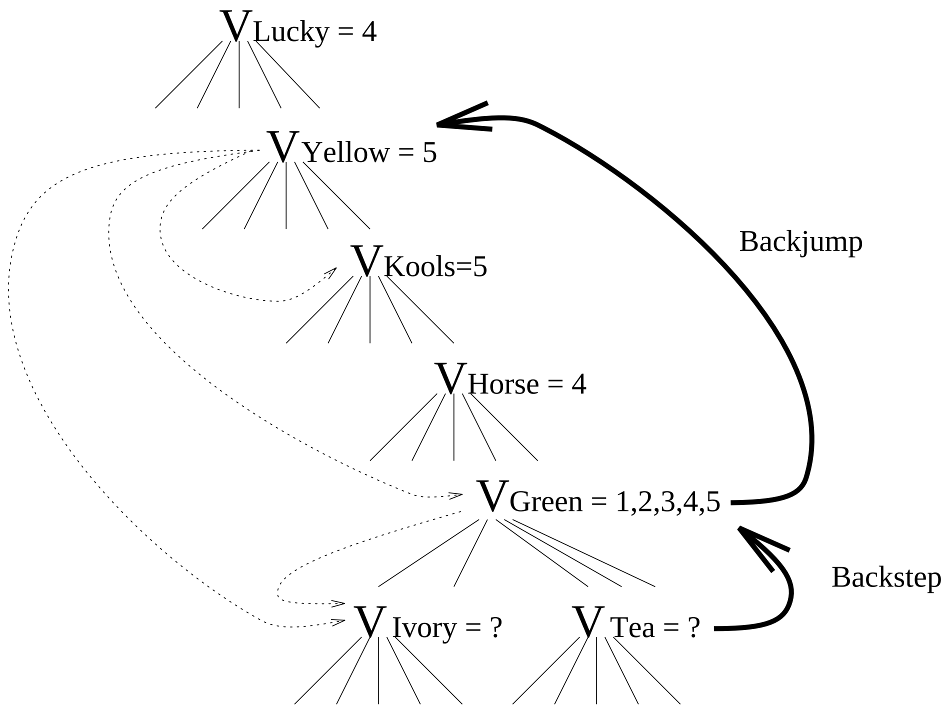


Figure 3.5: CBJvar scenario

node, and therefore only a backstep will occur. However, when all the values of the previous node have been tried a real backjump can occur to the deepest node in the union of the conflict sets of the two nodes. This backjump will prune a part of the search-tree and thus prevent performing some unnecessary checks. Some of the advantages of the CBJ backward move are therefore preserved under variable reordering.

Figure 3.5 shows an example of this behaviour. Instantiations of V_{Ivory} and V_{Tea} both fail due to conflicts with earlier variables. When all the values for V_{Green} are tried the algorithm backjumps to V_{Yellow} since this is the deepest variable with which V_{Green} had a conflict.

3.2.4 Backmarking with Variable Reordering (BMvar)

In [9] Prosser claims that Backmarking cannot exploit heuristic knowledge during the search process because it requires a static order of instantiation in order to maintain the integrity of its search knowledge (arrays *mcl* and *mbl*). This would suggest that BM, and the BM hybrids, cannot exploit heuristic knowledge during the search process. Prosser considered this to be a severe limitation on the worth of these algorithms. But, while incorporating value reordering heuristics in Backmarking might prove to be hard, this is certainly not the case with variable reordering. In [6] Haralick already used variable reordering with Backmarking and Forward Checking, he presented the results of a comparison between these two algorithms, although he did not include any actual code.

In order to correctly incorporate variable reordering in the BM algorithm, which uses a more informed forward move, the search information of the algorithm will also have to be stored using the extra indirection of the *ins* structure. Instead of accessing $mbl[i]$, we will now have to access $mbl[ins[i]]$, the maximum backup level of the i-th instantiation. The same indirection has to be used in the *mcl* structure. The modifications are shown in the *consistent* function of BMvar.

```
Function consistent(C, solution, current)
   NETWORK C;
   int current;
   SOLUTION solution;
{  int i;
   /* extra indirection for mbl and mcl                                */
   if (mcl[ins[current]][solution[ins[current]]] < mbl[ins[current]])
      return(0);
   for (i := mbl[ins[current]]; i < current; i++) { /* indirection mbl */
      mcl[ins[current]][solution[ins[current]]]:=i; /* indirection mcl */
      if (trivial(ins[current],ins[i])) continue;
         checks := checks + 1;
      if (C[ins[current]][ins[i]]                 /* extra indirection */
           [solution[ins[current]]][solution[ins[i]]] = 0)
         return(0);}
   return(1);
}
```

The same changes also have to be incorporated in the main function of BMvar. The *mbl* structure now has to be accessed by using the *ins*-array.

The boundaries of the final for-loop in which the *mbl* information is restored will also have to be changed. In case of a backtrack occurrence the *mbl*-array would formerly be updated from $current + 1$ to n. Since these boundaries no longer reflect the range of uninstantiated variables, this will have to be changed. We cannot use $ins[current + 1]$ because this value is still undefined when the algorithm has reached *current* (no variable has been assigned to spot $current + 1$ in the *ins*-array). To make sure that all uninstantiated variables are reset the algorithm will reset all $mbl[i]$ to the maximum of their current value and $current - 1$. Since none of the instantiated variables will have a *mbl* with a higher value it is guaranteed that no search knowledge about those variables will be disturbed.[2]

[2]If the *in_ins* function (see Section 3.2.1) is used the results are exactly the same.

```
Function BMvar(C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i;
   if (current = 1) {
      clear_setup(n, k);
      ins[1] := next(C,ins,solution,0,n,k);
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(1);
      else return(0);}
   for (i := 1; i <= k; i++) {
      if (C[ins[current]][ins[current]][i][i] = 0) continue;
      solution[ins[current]] := i;
      if (consistent(C, solution, current)){
         ins[current + 1] := next(C,ins,solution,current,n,k);
         if (BMvar(C, n, k, solution, current + 1, number, found))
            return(1); } }
   mbl[ins[current]] := current - 1;         /* extra indirection mbl */
   for (i := 1; i <= n; i++)                /* new boundaries for-loop */
      if (mbl[i] > current - 1)
         mbl[i] := current - 1;
  return(0);
}
```

In Figure 3.6 a BMvar scenario is shown. When the algorithm arrives at V_{Green} for the first time values 1, 2, 3 and 4 are found to be inconsistent with respectively V_{Red}, V_{Blue}, V_{Ivory} and V_{Yellow}. This information is stored in the *mcl* structure and the algorithm continues until a dead end or a solution is found. When a backtrack to V_{Ivory} occurs the order of the instantiation changes because V_{Green} now only has a single value left in its domain. By comparing the minimum backup level *mbl* with the values in the *mcl* structure the algorithm can make the two kinds of savings.

From this example we can see that the search information should be kept with

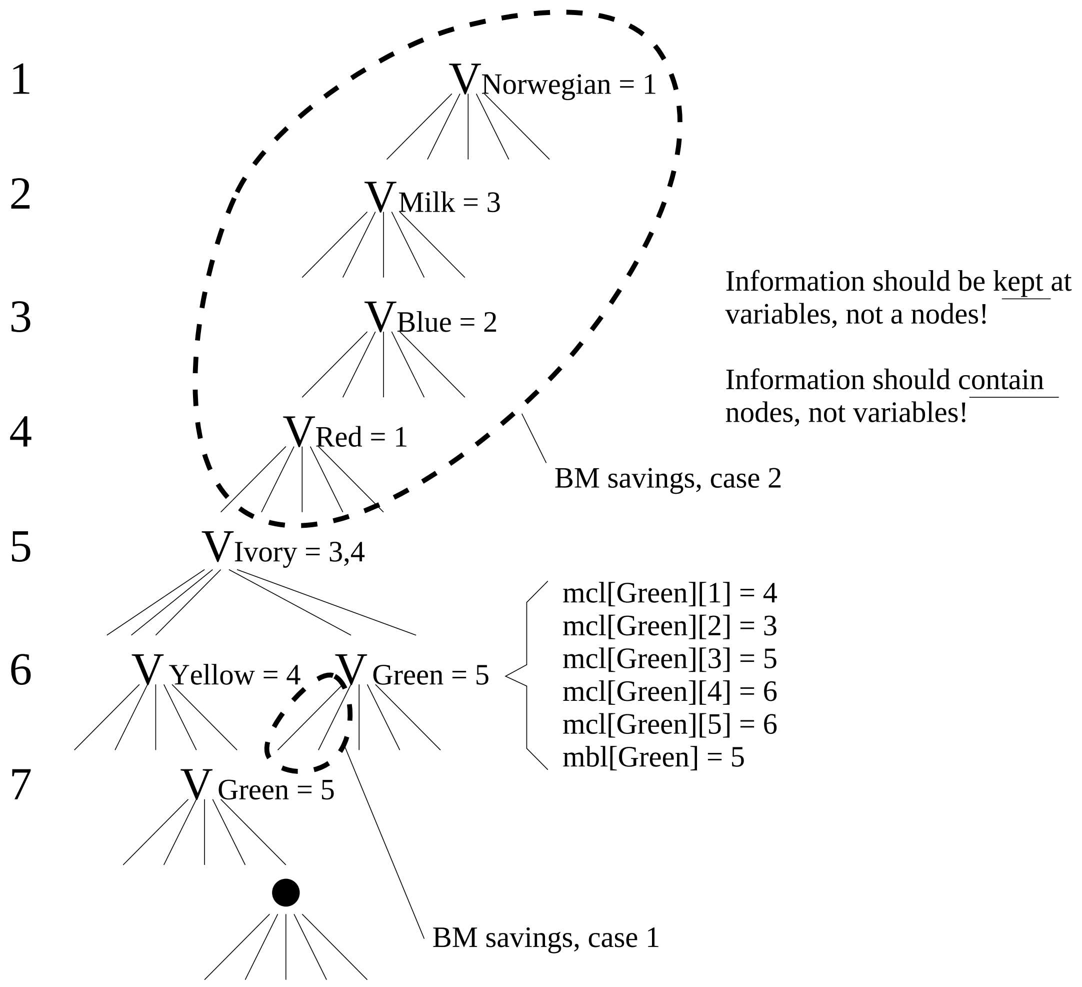


Figure 3.6: BMvar scenario

the variable, not with the nodes. This means that this information should be accessed through the *ins* structure. This information itself should contain nodes and not variables. If you compare this figure to Figure 1.5 the differences in the stored information can also be seen. In BM the *mcl* and *mbl* structures refered to specific variables, in BMvar they refer to nodes.

3.2.5 FC with Variable Reordering (FCvar)

Algorithms that involve Forward Checking can make use of a slightly different *next* function called *nextFC*. In FC algorithms there is no need to check the consistency of any of the remaining values of the current variable because inconsistent values where pruned in earlier processing. This advantage of FC algorithms over non-FC algorithms is the main contributer to a larger increase in performance from variable reordering for these algorithms. Haralick already stated that variable rearrangement improves forward checking more than backmarking because it has more information about the future [6].

The *FCnext* function is very similar to the original *next* function except for the absence of consistency checks and the use of the *domains*-structure to check if a domain value has been pruned earlier.

The *checking*-array in FC records the variables from which a particular variable prunes values. If $checking[i][j]$ was equal to one in the original FC algorithm this meant that variable i pruned values from variable j's domain and if $domains[j][l]$ was equal to i this meant that variable i pruned value l from variable j. In FCvar however, the algorithm forward checks an *instantiation*, which could be any variable, against all the future variables. So the first index in the *checking*-array needs an extra level of indirection and is changed to $ins[i]$. Note that this indirection can *not* be used for the second index because this index refers to an as yet uninstantiated variable.

```
Function nextFC (C,ins,domains,var,nvars,k)
   NETWORK C;
   int ins[N];
   int domains[N][K], var,nvars,k;
{  int count,i,j,l,m,nt,failed;
   min := k + 1;                                    /* initialize min */
   /* count values for all variables, break off if wipe-out found      */
   for(i:=1; (i <= nvars) and (min <> 0);i++) {
      if (not in_ins(i,ins,var)) {            /* not yet instantiated */
         count := 0;
         /* check all their values, break off if count>min             */
         for (j:=1; (j <= k) and (count < min);j++) {
            if (C[i][i][j][j] = 0) failed := true;    /* in the domain */
            else failed := false;
            if ((domains[i][j] = 0) and not failed)      /* not pruned */
               count := count + 1;}  /* another value left for this var*/
         if (count < min) {   /* is the count smaller than the minimum */
            min := count;                          /* record the new min */
            nt := i;} } }         /* set nt to number of this variable */
   return(nt);              /* return variable with fewest values left */
}
```

```
Function check_forward(C, k, i, j, value)
   NETWORK C;
   int k, i, j, value;
{  int m, old_count, delete_count;
   old_count := 0, delete_count := 0;
   for (m := 1; m <= k; n++)
      if (C[j][j][m][m] and domains[j][m] = 0) {
         old_count := old_count + 1;
         checks := checks + 1;
         if (C[ins[i]][j][value][m] = 0) {  /* indirection first index */
            domains[j][m] := i;
            delete_count := delete_count + 1; } }
   if (delete_count)
      checking[ins[i]][j] := 1;            /* indirection first index */
   return(old_count - delete_count);
}
```

In the *restore* function the algorithm can no longer assume that the variables $i+1$ to n are uninstantiated and therefore all the variables have to be checked. This changes the boundaries of the for-loop now to 1 and n. The auxiliary function $in\_ins(j, ins, i)$ checks if variable j is present in the list of instantiated variables. If this is not the case and values from the domain of this variable where pruned by i then these values are restored.

```
Function restore(i, n, k)
   int i, n, k;
{  int j, l;
   for (j := 1; j <= n; j++)                            /* new boundaries */
      if (not in_ins(j,ins,i) and             /* "not instantiated" check */
         (checking[ins[i]][j])) {
         checking[ins[i]][j] := 0;             /* indirection first index */
         for (l := 1; l <= k; l++)
            if (domains[j][l] = i)
               domains[j][l] := 0; }
}
```

The same explanation holds for the *consistent* function. The boundaries of the for loop change and an extra check is included to make sure that the algorithm does not forward check against already instantiated variables.

```
Function consistent(C, n, k, current, value)
   NETWORK C;
   int n, k, current, value;
{  int i;
   for (i := 1; i <= n; i++) {                          /* new boundaries */
      if (trivial(ins[current],i)) continue;
      if (not in_ins(i,ins,current)           /* "not instantiated" check */
         if (check_forward(C, k, current, i, value) = 0)
            return(0); }
  return(1);
}
```

The main FCvar function does not need any changes beside the obvious ones.

```
Function FCvar(C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i;
   if (current = 1) {
      clear_setup(n, k);
      ins[1] := nextFC(C,ins,domains,0,n,k);        /* first variable */
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(1);
      else return(0);}
   for (i := 1; i <= k; i++) {
      if (C[ins[current]][ins[current]][i][i] = 0 /* extra indirection */
         or domains[ins[current]][i])            /* extra indirection */
         continue;
      solution[ins[current]] := i;               /* extra indirection */
      /* select next variable using nextFC                            */
      if (consistent(C, n, k, current, solution[ins[current]])) }
         ins[current + 1] := nextFC(C,ins,domains,current,n,k);
         if (FCvar(C, n, k, solution, current + 1, number, found))
            return(1);}
      restore(current, n, k); }
  return(0);
}
```

In the Figure 3.7 an example of the behaviour of FCvar is shown. When $V_{Norwegian}$ is instantiated is prunes all but one value from the domain of V_{Blue} since constraint 15 states that "The Norwegian lives next to the Blue house". After the instantiation of V_{Blue} and V_{Milk}, V_{Red} is left with four possible value 1, 3, 4 or 5. Value 1 fails consistency checks since it would wipe out the domain of a future variable, $V_{Englishman}$ in this case, as a result of the constraint that states that "The Englishman lives in the Red house". This house is already occupied by the Norwegian.

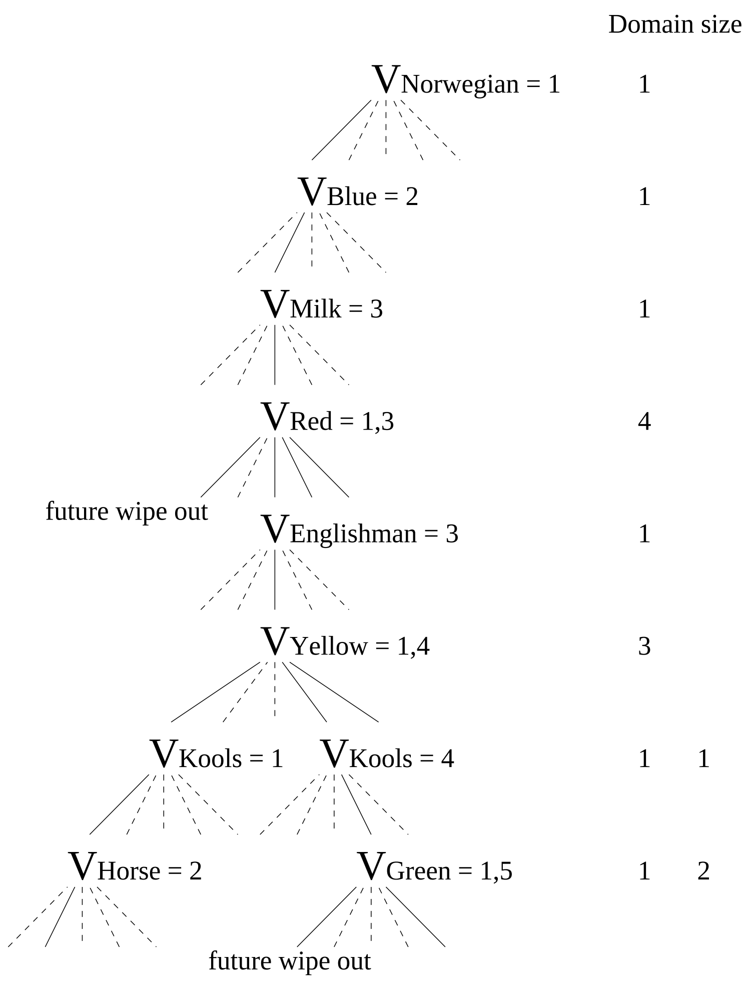


Figure 3.7: FCvar scenario

The search will continue along the branch with $V_{Englishman}$, V_{Yellow}, V_{Kools} and V_{Horse}, and find a solution deeper in the tree. When the algorithm backtracks looking for the next solution and instantiates V_{Yellow} with its next possible value and new branch will be formed. The ordering in this branch might be different from the previous one as we see happening in the example. After V_{Kools} it is V_{Green}, and not V_{Horse} that will be instantiated next. The cause of this change is constraint 12, "The Kools are smoked in the house next to the house with the Fox owner". A value of 1 for V_{Kools} will eliminate all but one value from the domain of V_{Horse}, a value of 4 leaves two possible domain values, and since V_{Green} appears earlier in the chronological ordering it will be selected first.[3]

3.3 Prosser's Hybrid Algorithms with Variable Reordering

Now that variable reordering versions of the five traditional CSP algorithms have been constructed the logical next step is to incorporate this heuristic into Prosser's hybrid algorithms. This results in four new algorithms: BMJvar, BM-CBJvar, FC-BJvar and FC-CBJvar (see Figure 3.8).

As shown in the discussion of the previous five algorithms the extra level of indirection is only needed for algorithms with a more informed forward move. Since Prosser's algorithms are hybrid, combining the forward move of one algorithm with the backward move of another, the extra indirection only has to be incorporated in the parts of these algorithms related to this forward move.

[3] V_{Yellow} and V_{Ivory} were swapped in the chronological ordering to simplify this example.

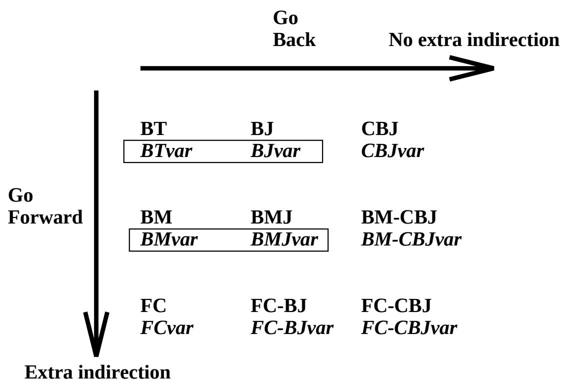


Figure 3.8: Variable reordering versions of Prosser's algorithms

3.3.1 Backmarking with Backjumping and Variable Reordering (BMJvar)

Previously we saw that algorithms with only a forward move need an extra level of indirection and algorithms with only a backward move do not. Since BMJ is a combination of BM, representing the forward move, and BJ, representing the backward move, and the information for these moves is not combined, BMJvar only needs the extra indirection for its BM part. Because of this the *ins*-array is only used in combination with *mcl* and *mbl* in the *consistent* function of this algorithm.

```
Function consistent(C, solution, current)
   NETWORK C; int current; SOLUTION solution;
{  int i;
   /* BM, extra indirection */
   if (mcl[ins[current]][solution[ins[current]]] < mbl[ins[current]])
      return(0);
   /* BM, extra indirection */
   for (i := mbl[ins[current]]; i < current; i++) {
      /* BM, extra indirection */
      mcl[ins[current]][solution[ins[current]]] := i;
      if (trivial(ins[current],ins[i])) continue;
      checks := checks + 1;
      if (C[ins[current]][ins[i]]
           [solution[ins[current]]][solution[ins[i]]] = 0) {
         if (i > jump_place[current])           /* BJ, no indirection */
            jump_place[current] := i;           /* BJ, no indirection */
         return(0); } }
   jump_place[current] := current - 1;          /* BJ, no indirection */
   return(1);
}
```

In the main function of BMJvar the information for both moves has to be restored when a backtrack occurs. As in BJvar restoring the *jump_place*-array does not give any problems and as in BMvar the boundaries for the for-loop in which *mbl* is restored need to be changed. The rest of the function is similar to the original BMJ function, except for the obvious changes.

```
Function BMJvar(C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i, jump;
   if (current = 1) {
      clear_setup(n, k);
      ins[1] := next(C,ins,solution,0,n,k);   /* select first variable */
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(0);
      else return(n);}
   for (i := 1; i <= k; i++) {
      if (C[ins[current]][ins[current]][i][i] = 0)/* extra indirection */
         continue;
      solution[ins[current]] := i;               /* extra indirection */
      if (consistent(C, solution, current)) {
         /* select next variable */
         ins[current + 1] := next(C,ins,solution,current,n,k);
         jump := BMJvar(C, n, k, solution, current + 1, number, found);
         if (jump <> current)
            return(jump); } }
   jump := jump_place[current];
   mbl[ins[current]] := jump;
   for (i := 1 ; i <= n; i++)                 /* BM, boundaries changed */
      if (mbl[i] > jump) mbl[i] := jump;
   for (i := jump+1; i <= current; i++)     /* BJ, boundaries unchanged */
      jump_place[i] := 0;
   return(jump);
}
```

BMJvar shows no improvements over BMvar just like BJvar shows improvements over BTvar. The variable reordering heuristic eliminates the usefulness of Backjumping because the heuristic will select a variable with an empty domain to be instantiated next. There will not be any instances where the source of a conflict is more than one position earlier in the search-tree and therefore all the backjumps will be backsteps. In Figure 3.8 the equivalence of BTvar-BJvar and BMvar-BMJvar is indicated by grouping them in the two small boxes.

3.3.2 Backmarking with Conflict-Directed Backjumping and Variable Reordering (BM-CBJvar)

The *union_conflicts* and *empty_conflicts* functions in BM-CBJ only involve information from the CBJ part of the algorithm and therefore they remain unchanged in BM-CBJvar.

```
Function consistent(C, solution, current)
   NETWORK C;  int current;  SOLUTION solution;
{  int i;
   /* BM, everywhere extra indirection for mcl and mbl                 */
   if (mcl[ins[current]][solution[ins[current]]] < mbl[ins[current]]) {
      conflicts[current][mcl[ins[current]][solution[ins[current]]]] := 1;
      if (conflicts[current][current] <
          mcl[ins[current]][solution[ins[current]]])
         conflicts[current][current] :=
          mcl[ins[current]][solution[ins[current]]];
      return(0);}
   for (i := mbl[ins[current]]; i < current; i++) {
      mcl[ins[current]][solution[ins[current]]] := i;
      if (trivial(ins[current],ins[i])) continue;
      checks := checks + 1;
      if (C[ins[current]][ins[i]]
            [solution[ins[current]]][solution[ins[i]]] = 0){
         conflicts[current][i] := 1;                /* CBJ, no change! */
         if (conflicts[current][current] < i)
            conflicts[current][current] := i;
         return(0); } }
  return(1);
}
```

The *consistent* function does change since it also uses information from the BM part of the algorithm. The *mbl* and *mcl* arrays get an extra level of indirection because they hold BM information, the *conflicts*-array does not change because it holds information for CBJ.

```
Function BM-CBJvar(C, n, k, solution, current, number, found)
   NETWORK C; int n, k, current, number, *found; SOLUTION solution;
{  int i,j, jump, curr;
   curr := count;
   if (current = 1) {
      clear_setup(n, k);
      ins[1] := next(C,ins,solution,0,n,k);
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(0);
      else return(n);}
   for (i := 1; i <= k; i++) {
      if (C[ins[current]][ins[current]][i][i] = 0) continue;
      solution[ins[current]] := i;
      if (consistent(C, solution, current)) {
         ins[current + 1] := next(C,ins,solution,current,n,k);
         jump := BM-CBJvar(C, n, k, solution, current+1, number, found);
         if (jump <> current){
            return(jump);} } }
   if (curr = count) jump := conflicts[current][current];
   else jump := current - 1;

   mbl[ins[current]] := jump;                /* BM, extra indirection */
   union_conflicts(jump, current);
   for (i := 1; i <= n; i++)                /* BM, boundaries changed */
      if (mbl[i] > jump) mbl[i] := jump;
   empty_conflicts(jump, current);
   return(jump);
}
```

The main function of BM-CBJvar is again very similar to BM-CBJ. Just like in previous algorithms with a combination of BM and variable reordering the boundaries on the final for-loop have to be changed to ensure that all elements of *mbl* are updated correctly.

3.3.3 Forward Checking with Backjumping and Variable Reordering (FC-BJvar)

The *check_forward* and *restore* functions of FC-BJvar are the same as those for FCvar since they only deal with the FC part of the algorithm. The *consistent* function is changed the same way the FCvar *consistent* function changed: the boundaries of the for-loop are changed and an extra test is included to make sure that the variable that is forward-checked against is not already instantiated.

```
Function consistent(C, n, k, current, value)
   NETWORK C;
   int n, k, current, value;
{  int i;
   for (i := 1; i <= n; i++) {               /* FC, boundaries changed */
      if (trivial(ins[current],i)) continue;
      if (not in_ins(i,ins,current))       /* not already instantiated */
        if (check_forward(C, k, current, i, value) = 0)
           return(i);}
  jump_place[current] := current - 1;
  return(0);
}
```

The main function of FC-BJ is changed in the same way. The *conflicts*-array, belonging to the FC part of the algorithm, gets an extra level of indirection for its first index. The *jump_place*-array does not change since it belongs to the BJ part of the algorithm. The boundaries of the for-loops in this function do not have to be changed because they deal with past variables, from 1 to *current* which are guaranteed to be represented in the *ins*-array.

```
Function FC-BJvar (C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i, j, jump, fail;
   if (current = 1) {
      clear_setup(n, k);
      ins[1] := nextFC(C,ins,domains,0,n,k);
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(0);
      else return (n);}
   for (i := 1; i <= k; i++) {
      if (C[ins[current]][ins[current]][i][i] = 0 or
          domains[ins[current]][i]) continue;
      solution[ins[current]] := i;
      fail := consistent(C, n, k, current, solution[ins[current]]);
      if (not fail) {
         ins[current + 1] := nextFC(C,ins,domains,current,n,k);
         jump := FC-BJvar(C, n, k, solution, current + 1, number, found);
         if (jump <> current) return(jump); }
      restore(current, n, k);
      if (fail)
      for (j := 1; j < current; j++)
         if (checking[ins[j]][fail] and jump_place[current] < j)
            jump_place[current] := j; }
   jump := jump_place[current];
   for (i := 1; i <= current; i++)
      if (checking[ins[i]][ins[current]] and jump < i)
         jump := i;
   for (i := current; i > jump; i--) {
      jump_place[i] := 0;
      restore(i, n, k);}
   return(jump);
}
```

In contrast to the equality of BMJvar and BMvar, Forward Checking with Backjumping and variable reordering (FC-BJvar) does have some, albeit limited, benefits over FCvar because in this algorithm the backjump information is used in a different way. The *jump_place*-array is updated when a wipe-out has occurred in the future and its information is then used to jump back to the cause of the conflict (see Figure 2.3).

3.3.4 Forward Checking with Conflict-Directed Backjumping and Variable Reordering (FC-CBJvar).

The $union\_conflicts$ and $empty\_conflicts$ functions do not change since they only handle information from the FC part of the algorithm. The functions *check_forward*, *restore* and *consistent* are the same as in FCvar for the same reasons. The main function is changed in the usual way, the *checking*-array gets an extra level of indirection for its first index and the trivial changes are make to the C-matrix.

```
Function FC-CBJvar(C, n, k, solution, current, number, found)
   NETWORK C;
   int n, k, current, number, *found;
   SOLUTION solution;
{  int i, jump, fail, curr;
   curr := count;
   if (current = 1) {
      clear_setup(n, k);
      ins[1] := nextFC(C,ins,domains,0,n,k);
      *found := 0;}
   else if (current > n) {
      process_solution(C, n, solution);
      *found := 1;
      count := count + 1;
      if (number = 1) return(0);
      else return (n);}
   for (i := 1; i <= k; i++) {
      if (C[ins[current]][ins[current]][i][i] = 0 or
          domains[ins[current]][i]) continue;
      solution[ins[current]] := i;
      fail := consistent(C, n, k, current, solution[ins[current]]);
      if (not fail) {
         ins[current + 1] := nextFC(C,ins,domains,current,n,k);
         jump := FC-CBJvar(C, n, k, solution, current+1, number, found);
         if (jump <> current) return(jump); }
      if (fail) union_checking(current, fail);
      restore(current, n, k);}
   if (curr =  count) {
      jump := 0;
      for (i := 1; i < current; i++)
         if (conflicts[current][i]) jump := i;
      for (i := jump + 1; i < current; i++)
         if (checking[ins[i]][ins[current]]) jump := i;  }
   else jump := current - 1;
   union_checking(current, current);
   union_conflicts(jump, current);
   for (i := current; i > jump; i--) {
      empty_conflicts(i);
      restore(i, n, k);}
   restore(jump, n, k);
   return(jump);
}
```

The *union_checking* function combines the information from the *conflicts*-array, from the CBJ part of the algorithm, with the *checking*-array of the FC part. Because this function is called with two different sets of parameters, *(current,fail)* and *(current,current)* a few additional changes have to be made to the original version. If the function is called with *(current,fail)* the second parameter represents a variable that is not yet instantiated. If it is called with *(current,current)* the second parameter represent a variable that is instantiated and thus has to be accessed through the *ins*-array.

```
Function union_checking(i,j)
   int i, j;
{  int m;
  if (not in_ins(j,ins,i)) {      /* is second parameter instantiated ? */
     for (m := 1; m < i; n++)
         /* j is not instantiated */
         conflicts[i][m] := conflicts[i][m] or checking[ins[m]][j];}
  else {
     for (m := 1; m < i; n++)
        /* j is instantiated */
        conflicts[i][m] := conflicts[i][m] or checking[ins[m]][ins[j]];}
}
```

3.4 Other heuristics

3.4.1 Value Reordering

Another heuristic that could be used to improve CSP algorithms is value reordering. A value reordering heuristics selects the value from a variables domain that should be used for the instantiation of the current variable. Some values will be more restrictive for the future search than others and thus result in different search-trees. One heuristic attempts to select the value that least restricts the future search. This way there will be a greater chance of finding a solution on the current search-path.

Value reordering is only useful when we are looking for the first k solutions to a CSP. If we are looking for all solutions then all values for a particular variable will have to be checked, which makes the ordering of the values unnecessary. The same holds for a CSP that does not have any solutions, in order to determine this the entire tree will have to be searched and value reordering does not have any advantages. These two disadvantages pose a severe restriction on the use of value reordering heuristics.

In [7] Kalé uses value rearrangement together with variable rearrangement to find solutions for the n-Queens problem for all values of n from 4 to 1000. This heuristic appears to be almost perfect in the sense that it finds a first solution without any backtracks in most cases.

For value ordering heuristics we can again use both global and local orderings. A global ordering is an ordering of the variables before the search starts and a local ordering is an ordering that takes place during the search. A local value ordering is not necessarily better than a global ordering. In the n-Queens problem for instance we can use a very simple global ordering by instantiating the variables from "the inside out". This implies that we try values in the middle of a row first and then move outward to the edges of the board. This heuristic works a lot better than one level local value reordering in simple Backtracking because a local heuristic returns the same values for all but the outer two squares. In order to break the ties between all the identical values a local algorithm will have to look deeper into the search tree. Because the n-Queens problem is a well-structured problem we can tell in advance which ordering will result from this: "the inside out" order.

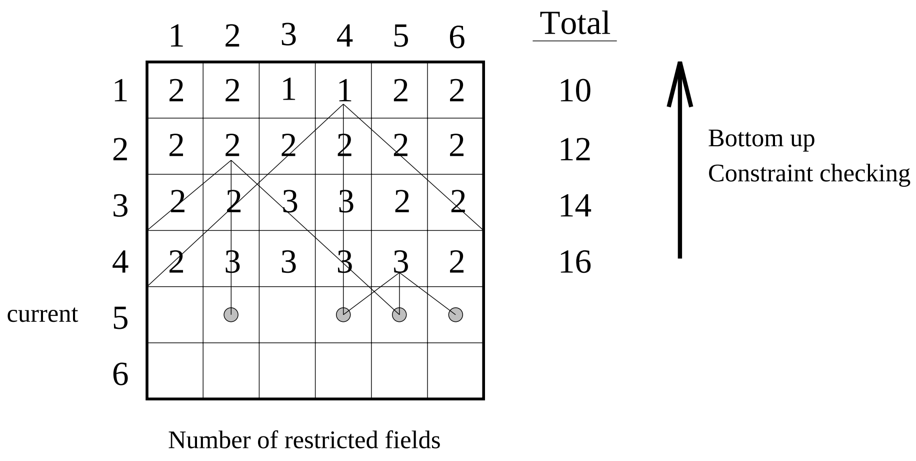


Figure 3.9: Global Constraint Reordering

3.4.2 Constraint Reordering

The order in which the constraints are checked can also have great impact on the performance of the algorithm. If a new variable i in instantiated only a subset of the total set of constraints has to be checked, this set contains only those constraints that involve the newly instantiated variable. As soon as one of these constraints in this subset is violated we know that we can abandon the current search path due to the nature of our search. To improve the search we therefore desire to find a conflicting constraint as early as possible. This approach leads us to an order in which the most rigid constraints are checked first. The checking order can again be determined globally or locally. Although the actual constraints do not change the subset relevant to the instantiated variables does.

A good global ordering in the n-Queens problem would for example be to check the constraints "bottom-up" (assuming a top-down instantiation of the rows) starting from the previous row. A Queen that is placed close to the current row constraints the possible values for this row more than a Queen that is placed further away from it. A Queen in an adjacent row gives $3*(n-2)+2*2$ possible conflicts, the two corner squares each restrain two squares, the $n-2$ other placements each result in 3 possible conflicts. The general formula for the number of possible conflicts is $3n-2i+2j$ with $j<i$, this formula decreases when i and j are further apart (Figure 3.9).

3.5 Summary

If we include the variable reordering versions of the algorithms into our summary we get table 3.1.

Algorithm	forward move	backward move	next variable
BT	check against all past variables	previous variable	chronological
BJ	check against all past variables	single jump back	chronological
CBJ	check against all past variables	multiple jumps back	chronological
BM	perform only new checks	previous variable	chronological
FC	prune future variables	previous variable	chronological
BMJ	perform only new checks	single jump back	chronological
BM-CBJ	perform only new checks	multiple jumps back	chronological
FC-BJ	prune future variables	single jump back	chronological
FC-CBJ	prune future variables	multiple jumps back	chronological
BTvar	check against all past variables	previous variable	least values
BJvar	check against all past variables	single jump back	least values
CBJvar	check against all past variables	multiple jumps back	least values
BMvar	perform only new checks	previous variable	least values
FCvar	prune future variables	previous variable	least values
BMJvar	perform only new checks	single jump back	least values
BM-CBJvar	perform only new checks	multiple jumps back	least values
FC-BJvar	prune future variables	single jump back	least values
FC-CBJvar	prune future variables	multiple jumps back	least values

Table 3.1: Summary of all algorithms

Chapter 4

Results

4.1 Estimating the cost of CSP algorithms

To estimate the cost of a specific instance of a backtrack search tree we can use an approach by Knuth [8]. Instead of giving a mathematical formula Knuth uses a Monte Carlo approach to predict the number of nodes in a search tree when looking for all solutions. This approach is based on a random exploration of the search tree. For each partial solution $(v_1, \ldots, v_k)$ for $0 \leq k < n$ a value for v_{k+1} is chosen from among the set of all possible continuations. By taking into account the number of possibilities at every level that are encounter during the random walk, an estimate for the total cost of the search can be computed. The expected value of this computed cost is proven to be equal to the cost of the tree. However, just knowing that executing the algorithm yields the right expected value is not very useful in practise. Therefore it is always necessary to use a number of trials to come up with a reasonable estimate. Purdom gave a number of improvements on Knuth's algorithm [10], as he found it to be ineffective in certain cases. His partial backtracking results in exponential improvements over Knuth's algorithm for estimating the tree size, by occasionally following more than one path from a node. This effect is particularly important for trees with a lot of dead-end branches (slender trees) and for tall trees.

The use of this Monte Carlo approach is mathematically well defined and this might suggest that it should also be possible to derive a single closed form formula of the same scope and accuracy. However, such a formula might be rather useless if its parameterization is not rich enough. The mean performance of the backtrack algorithm in solving all csp's with N variables might not be a very informative number because it ranges over a vast area of problems, from "n-Queens" and "Zebra-problems" to "instant insanity" and "uncrossed knight tours". This stands in contrast to, e.g., sorting where a one parameter formula is usually sufficient to describe the average behavior of an algorithm [5]. Therefore we need more problem specific parameters to distinguish one problem from all the others in the domain of the algorithm, and the need for computational models arises. Two examples of these models are:

Model 1 Purdom and Brown derived an asymptotic expression for the number of nodes in a tree constructed by a backtracking algorithm that finds all the solutions of conjunctive normal form formulas. These formulas range over v variables with s literals per term and v^{α} terms ($1 < \alpha < s$). The number of nodes is the backtrack tree will have is then: $e^{\Theta(v^{(s-\alpha)/(s-1)})}$.[1]

This shows that, since exhaustive search requires time $e^{\Theta(v)}$ and $(s-\alpha)/(s-1) < 1$, simple backtracking saves an exponential amount of time but still has exponential complexity. This set of problems had been selected because

[1]$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$

they lead to a relatively straightforward analysis. However, these problems are unlike typical problems that are solved by backtracking. Care is therefore needed in interpreting the results of this analysis [16].

Model 2 Haralick and Elliot [6] assume that a given pair of variables with a given pair of values are consistent with a fixed probability p, where p is independent of which variables or values have already been used in prior instantiations. They also assume that each variable has the same number of possible values M. Using these assumptions they arrive at:

$$\sum_{k=1}^{N}(M^k p^{(k-1)(k-2)/2}\frac{1-p^{k-1}}{1-p})$$

for the total estimated number of consistency checks in a BT algorithm.

Unfortunately both models rely heavily on very specific assumptions that are quite restrictive. Since backtracking is the simplest of the algorithms discussed in this thesis no attempt is made to derive analytical expressions for the more complex other algorithms. Instead we employ an empirical analysis.

4.2 Empirical results

4.2.1 Zebra problem

In order to compare the variable reordering versions of the hybrid algorithms with the originals by Prosser [9] we use the same problem he did to obtain our empirical results. In his paper Prosser uses instances of the Zebra problem (see Section 1.2.1) with different bandwidths.

The bandwidth of a variable v in a constraint graph under an ordering h is the maximum value of $\mid h(v)-h(w) \mid$ over all variables w connected to v. The bandwidth of a graph under an ordering is the maximum bandwidth of any variable [21]. To get a sizable test set 450 different orderings were used, 50 orderings of each of 9 different bandwidths between 16 and 24. The following results were obtained using the same orderings Prosser used.

In table 4.1 the first column shows the 18 different algorithms that were tested. The second column shows the total number of consistency checks needed by these algorithms to find the first solution for the 450 different orderings of the Zebra problem and the third column shows Prosser's results as a comparison. The last column shows the total number of consistency checks needed to find all 11 solutions to the problem.

The first observation we can make from table 4.1 is that the results confirm that the author's recursive algorithms are indeed functionally equivalent to Prosser's non-recursive algorithms, since they result in exactly the same number of average checks over 450 problems. Furthermore, it also confirms the earlier claim that BTvar and BJvar will result in the same number of checks because no backjumps will occur in a variable reordering version of BJ. The same can be said about BMvar and BMJvar which also result in the exact same number of checks.

In the table we can also see that all the variable reordering versions of the tree-search algorithms BT, BJ, CBJ, BM, BMJ and BM-CBJ result in approximately the same number of checks, around 22,000. Apparently the added value of combining the forward and backward moves of different algorithms diminishes when variable reordering heuristics are implemented in these combinations.

As expected the Forward Checking algorithms benefit the most from the heuris-

	First solution		All solutions	Ratio
Algorithm	van Run	Prosser	van Run	one - all
BT	3,858,988.8	3,858,989	16,196,383.9	4.2
BTvar	22,418.9		153,970.1	7.0
BJ	503,324.3	503,324	2,225,701.2	4.4
BJvar	22,418.9		153,970.1	7.0
CBJ	63,211.9	63,212	397,341.7	6.3
CBJvar	22,278.4		153,243.3	6.9
BM	396,944.9	396,945	1,607,386.7	4.0
BMvar	22,191.7		151,394.1	6.8
BMJ	125,473.9	125,474	557,188.8	4.4
BMJvar	22,191.7		151,394.1	6.8
BM-CBJ	25,470.5	25,470	160,113.6	6.3
BM-CBJvar	22,063.8		150,747.9	6.8
FC	35,582.0	35,582	181,984.7	5.1
FCvar	503.7		2,928.2	5.8
FC-BJ	16,839.5	16,839	101,823.9	6.0
FC-BJvar	502.9		2,924.9	5.8
FC-CBJ	10,361.2	10,361	69,982.1	6.8
FC-CBJvar	502.3		2,921.2	5.8

Table 4.1: Constraint checks, first and all solutions

tic and the difference between them and the tree-search algorithms becomes larger. Although BM-CBJ was better than FC in the original versions, the variable reordering version FCvar surpassed BM-CBJvar by two orders of magnitude. Furthermore there seems to be no clear split in the Forward Checking algorithms between FCvar on one side and FC-BJvar and FC-CBJvar on the other. Just like with the tree-search algorithms, combining the forward and backward moves of two algorithms does not seem to pay off.

We can now order the algorithms again according to the average number of checks they perform on this problem.

BT $>$ BJ $>$ BM $>$ BMJ $>$ CBJ $>$ FC $>$ (BTvar $=$ BJvar) $>$ CBJvar $>$ (BMvar $=$ BMJvar) $>$ BM-CBJvar $>$ FC-BJ $>$ FC-CBJ $>$ FCvar $>$ FC-BJvar $>$ FC-CBJvar

There seem to be no significant other observations that can be made by looking at the number of checks to find all solutions. The ordering of the algorithms stays the same, and the ratio for the difference in number of checks to find one solution and to find all solutions only ranges from 4.0 for BM to 7.0 for BTvar and BJvar. The only algorithms for which this ratio improves after the variable reordering heuristic is added are FC-BJ and FC-CBJ. These ratios might prove to be very problem specific, so their value should not be overrated.

Table 4.2 shows how often one algorithm (row) performed better than another (column) in the 450 different problems. Not performing better does not mean performing worse because in many cases the results from two algorithms were exactly the same. If we compare the table to the one in [9] it is again confirmed that the author's algorithms are equivalent to Prosser's as the tables match each other on

	BT	BT	BJ	BJ	CBJ	CBJ	BM	BM	BMJ	BMJ	BM-	BM-	FC	FC	FC-	FC-	FC-	FC-
		var		var		var		var		var	CBJ	CBJ		var	BJ	BJ	CBJ	CBJ
												var				var		var
BT	-	16	0	16	0	16	0	16	0	16	0	16	0	0	0	0	0	0
BTvar	434	-	369	0	217	0	340	0	272	0	135	0	193	0	111	0	63	0
BJ	450	81	-	81	0	81	132	80	0	80	0	80	0	0	0	0	0	0
BJvar	434	0	369	-	217	0	340	0	272	0	135	0	193	0	111	0	63	0
CBJ	450	233	450	233	-	233	370	232	280	232	0	232	130	0	35	0	5	0
CBJvar	434	211	369	211	217	-	340	14	272	14	135	0	193	0	111	0	63	0
BM	450	110	318	110	80	110	-	109	31	109	8	109	13	0	5	0	2	0
BMvar	434	442	370	442	218	428	341	-	273	0	138	61	195	0	112	0	63	0
BMJ	450	178	450	178	170	178	419	177	-	177	17	177	29	0	7	0	3	0
BMJvar	434	442	370	442	218	428	341	0	273	-	138	61	195	0	112	0	63	0
BM-CBJ	450	315	450	315	450	315	442	312	433	312	-	312	286	1	117	0	35	0
BM-CBJvar	434	442	370	442	218	442	341	138	273	138	138	-	195	0	112	0	63	0
FC	450	257	450	257	320	257	437	255	421	255	163	255	-	1	0	1	0	1
FCvar	450	450	450	450	450	450	450	450	450	450	449	450	449	-	446	0	444	0
FC-BJ	450	339	450	339	415	339	445	338	443	338	333	338	438	4	-	4	0	4
FC-BJvar	450	450	450	450	450	450	450	450	450	450	450	450	449	8	446	-	444	0
FC-CBJ	450	387	450	387	445	387	448	387	447	387	415	387	440	6	388	6	-	5
FC-CBJvar	450	450	450	450	450	450	450	450	450	450	450	450	449	14	446	6	445	-

Table 4.2: How often is one algorithm (row) better than another (column)

similar positions. The following conclusions can be made regarding these results.

Tree-search algorithms seem to have a declining rate of outperforming their originals. The better the original algorithm was, the smaller the chance that its variable reordering version outperforms the original. The rate ranges from 434/450 = 96% for BT to 138/450 = 31% for BM-CBJ. The mutual differences between related algorithms also seem to get smaller, for example in contrast to the relation between BMJ and BM-CBJ, BM-CBJvar is not often much better than BMJvar.

The scenario described in Chapter 2 in which BM outperforms BMJ and BM-CBJ must be rather rare because both BMJ and BM-CBJ outperform BM in more than 93% of the cases.

CBJvar only outperforms CBJ in 217 cases, in 233 cases CBJ is better and there are no cases in which they perform the same. Since CBJvar still outperforms CBJ in total by a factor of about 3, this means that the margin by which CBJvar outperforms CBJ in those cases must be significantly larger than the margin by which it performs worse in the other cases. The same can be said about BM-CBJvar, which only outperforms BM-CBJ in 138 cases and is worse in 312 cases. However, since the difference between the total values of these two algorithms in table 4.1 is rather small (only 15%), this is less of a surprise.

The forward checking algorithms with variable reordering show a significantly larger rate of outperforming their originals. FCvar, FC-BJvar and FC-CBJvar all perform better then their non-reordering counterparts in more than 444 out of 450 cases. However, FC-BJvar and FC-CBJvar only outperform FCvar in respectively 8 and 12 cases and in all other cases they perform the same. Similarly FC-CBJvar is only better than FC-BJvar in 6 cases and equal in all the others.

4.2.2 N-Queens problem

The n-Queens problem is often used by researchers in the AI-community as a benchmark for their programs because it can be solved by algorithms and heuristics that are widely applicable in other constraint-based optimization problems very common to computing practice. The results of the use of a standard problem to compare different CSP-algorithms must be handled with care though because they only reflect the behavior of the algorithms in one particular case. The general CSP is a very richly parameterized problem, and an algorithm that performs well for a problem like n-Queens with very specific characteristics, like $N = K$ and a complete graph representing its constraint matrix, can perform very differently on a problem with different characteristics.

n-Queens problem: place n-Queens on a $n \times n$ chess board in such a way that neither Queen attacks another one. Since every row can only hold one Queen, the problem is usually transformed to finding a position on every row of a chess board for one Queen without them attacking each other.

Again we can be interested in the first, any, or all solutions to the problem. Two examples of solutions for $n = 4$ and $n = 8$ are given in Figure 4.1, the latter being one of 92 possible solutions to this particular problem.

Although finding one explicit solution to the n-Queens problem can be solved analytically [1], finding the first, any or all solutions cannot, and thus offers an interesting benchmark for search algorithms.

In Figure 4.2 and 4.3 the graphs are displayed for the number of checks needed to find the first and all solutions to the n-Queens problem. The algorithms used are

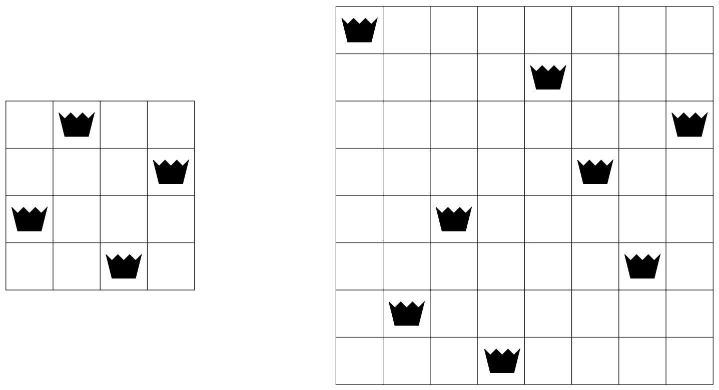


Figure 4.1: A solution to the n-Queens problem for n=4 and n=8

the best ones from the previous section: BM-CBJvar, FC, FCvar and FC-CBJvar. FC-BJvar is not used because it behaves almost identically to FC-CBJvar.

In Figure 4.2 we can see that FC now performs worse than BM-CBJvar (and very likely also the other tree-search hybrids) based on the number of consistency checks it needs to find the first solution. The n-Queens problem has a very dense (complete) constraint matrix and since there are no trivial constraints FC has to forward check against every other variable. This results in a larger number of consistency checks. The FC hybrids that were combined with the variable reordering heuristic do remain faster than the tree-search hybrids. However, there seems to be no important advantage from the use of hybrid FC algorithms with variable reordering (e.g., FC-CBJvar) over variable reordering with the standard FC algorithms (FCvar) since the graphs for these two algorithms coincide.

Figure 4.3 shows that BM_CBJvar needs considerably more consistency checks to find all solutions to the n-Queens problems than FC. The advantage it had in finding the first solution seems to have disappeared. While the overhead of the FC algorithm proves to be the determining negative factor in the case with the first solution, it pays off when the search continues for more solutions. FCvar and

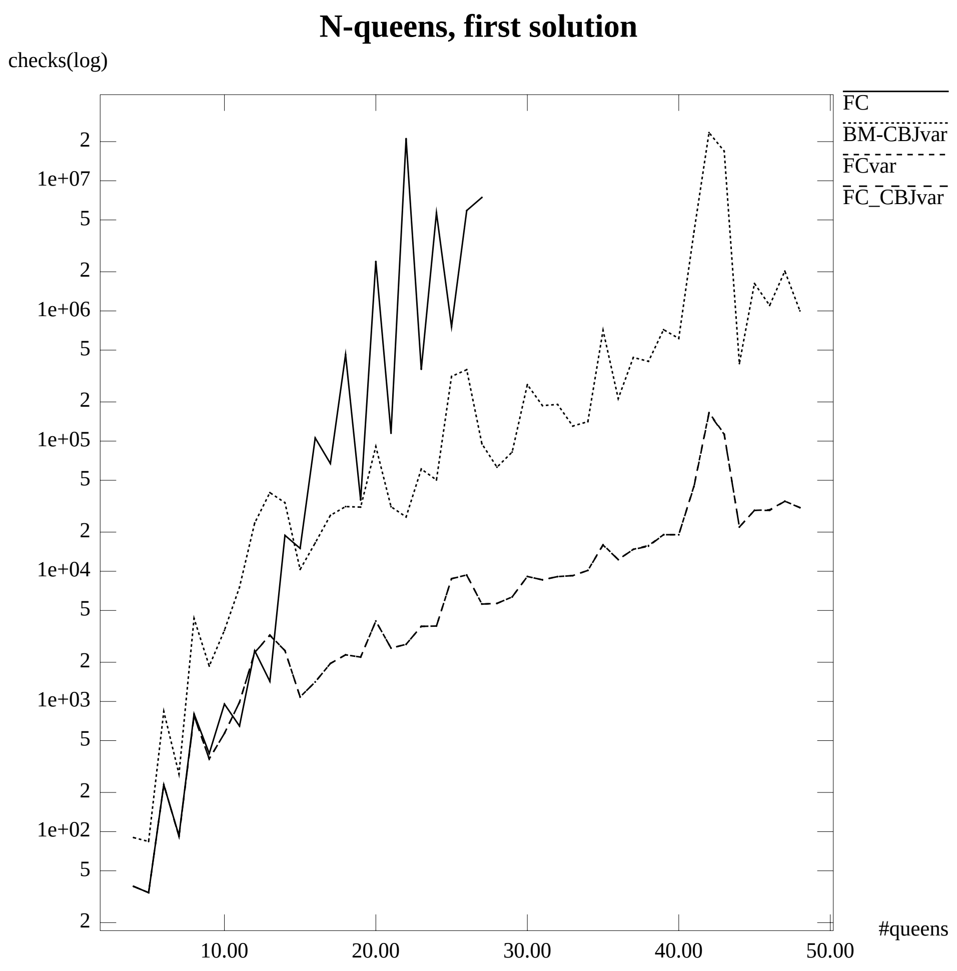


Figure 4.2: N-Queens, first solutions

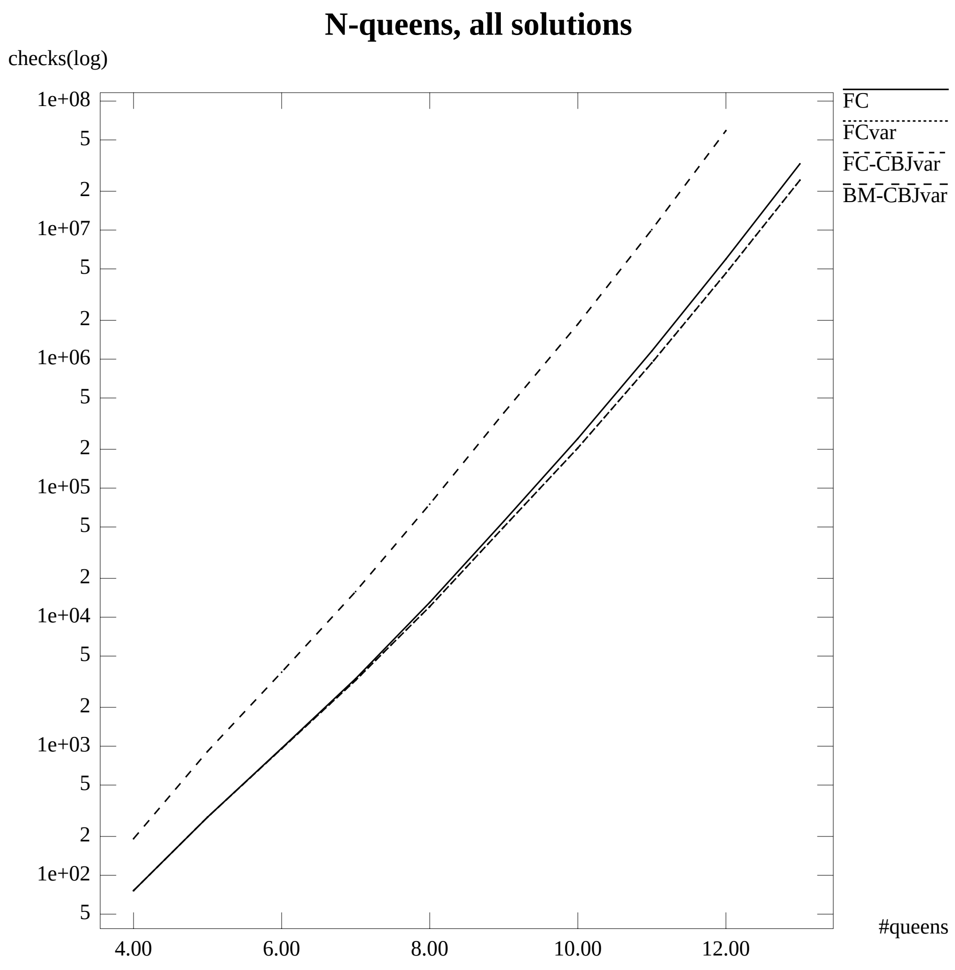


Figure 4.3: N-Queens, all solutions

FC-CBJvar coincide again but they form a consistent improvement of performance over FC.

4.2.3 Random problems

Both the n-Queens and the Zebra problem have a very specific structure. In the n-Queens problem the constraint matrix represents a complete graph, there is a constraint between every pair of Queens. The Zebra problem also has a very distinctly shaped graph with variables grouped in five hexagons. The variables in each hexagon are highly constraint among themselves and loosely constraint with variables from other hexagons (see Figure 1.1). The third problem used for testing has therefore a more randomly constructed constraint matrix.

The problem has 25 variables (n), each with 15 different values (k). The constraint matrix was build randomly using two parameters p and q, where p is the independent probability that a constraint between two variables is not trivial and q is the independent probability of a 1 as an entry in a non-trivial constraint.

The results are obtained by taking the average number of checks needed to find the first solution in 10 problems with a specific p and q. Tests are performed for p and q ranging from 0 to 100 with step size 10. All problems are guaranteed to have at least one solution because one solution was put into every matrix in a random fashion.

The three best algorithms discovered so far were subjected to this test, FCvar, FC-BJvar, and FC-CBJvar.

FC-CBJvar, FC-BJvar and FCvar demonstrated very similar behavior and since putting more results in the same graph would make it difficult to read only a graph for FC-CBJvar is given. As we can see in Figure 4.4 (logarithmic) the highest

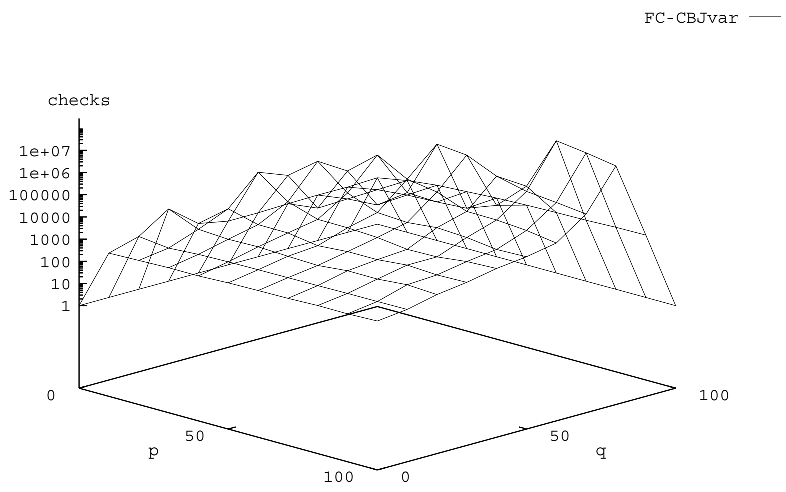


Figure 4.4: Checks for FC-CBJvar

number of checks are needed for problem with p between 50 and 100 p and q between 70 and 80. A higher number of checks for a higher value of p is predictable, more non-trivial constraints lead to more failures and thus to more checks. The same can be said for the low number of checks for $q = 90$ or $q = 100$, since these values of q turn the non-trivial constraints into almost trivial or trivial ones. The peak at q between 70 and 80 can be explained by looking at $1/q$ as a cutoff probability. When q is low a high number of branches in the search tree are cut off early. When q grows larger and the cutoff probability decreases it takes the algorithm longer to identify a branch as being unsuccessful.

In Figure 4.5 (linear) the differences are shown between FCvar and FC-BJvar. The first observation we can make is that FCvar never outperforms FC-BJvar, this corresponds to what we saw in the previous two problems. FC-BJvar outperforms FCvar in a few cases, especially in the case when p is between 80 and 100 and

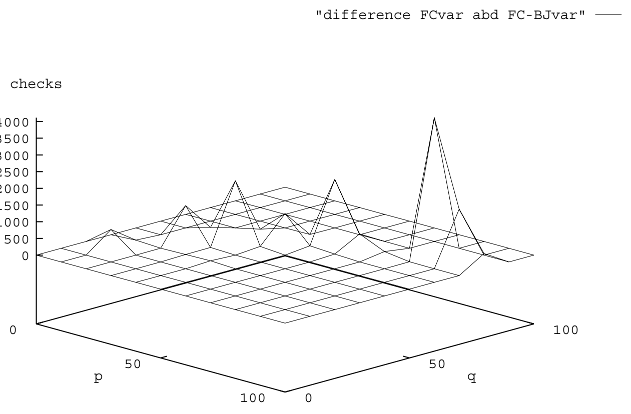


Figure 4.5: Difference between FCvar and FC-BJvar

$q \approx 80$. This is the area in which all algorithms need the largest number of checks and these difficult problems seem to result in relatively larger gains for FC-BJvar.

The differences between FC-CBJvar and FC-BJvar are shown in Figure 4.6 (linear) and they are of a larger order than those between FC-BJvar and FCvar (notice the difference in scale on the *checks* axis). The differences between FC-BJvar and FC-CBJvar also seem to be more scattered although the peak at $p = 80$ and $q = 80$ is in accordance with what we saw in the previous comparison: the more complex hybrids perform better on the more difficult problems. The rest of the differences lie mostly around $1/2 < p/q < 1$. Even though the difference between the two algorithms might seem large for these particular problems it still represents a difference of less than 1% in the total number of checks.

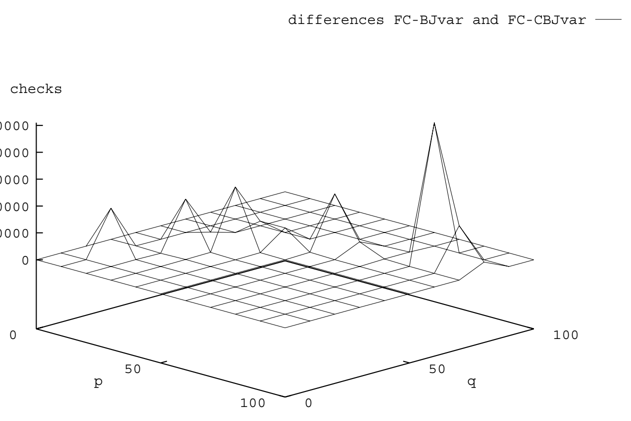


Figure 4.6: Difference between FC-BJvar and FC-CBJvar

4.3 Influence of Uniform Domain sizes

The loss of effectiveness of the hybrid algorithms can possibly be explained by the uniformity of the domain sizes. The Zebra problem, the n-Queens problem and the random problems all have variables with fixed domain sizes. The variable reordering heuristic that was used tends to select variables that have been filtered by past variables. This causes the source of a conflict to be close to its occurence and therefore the effect of the backward move diminishes. When the initial domain sizes have larger size differences the hybrid algorithms might regain some of their value.

In Figure 4.7 an example of a problem with non-uniform domain sizes is given in which FC-CBJvar would retain more of its benifits compared to the problems that were used previously. The large domain of V_m gets wiped out by a combination of variables that are not located closely together in the search tree. Therefore the CBJ backward move of the algorithm enables it to jump back using large jumps,

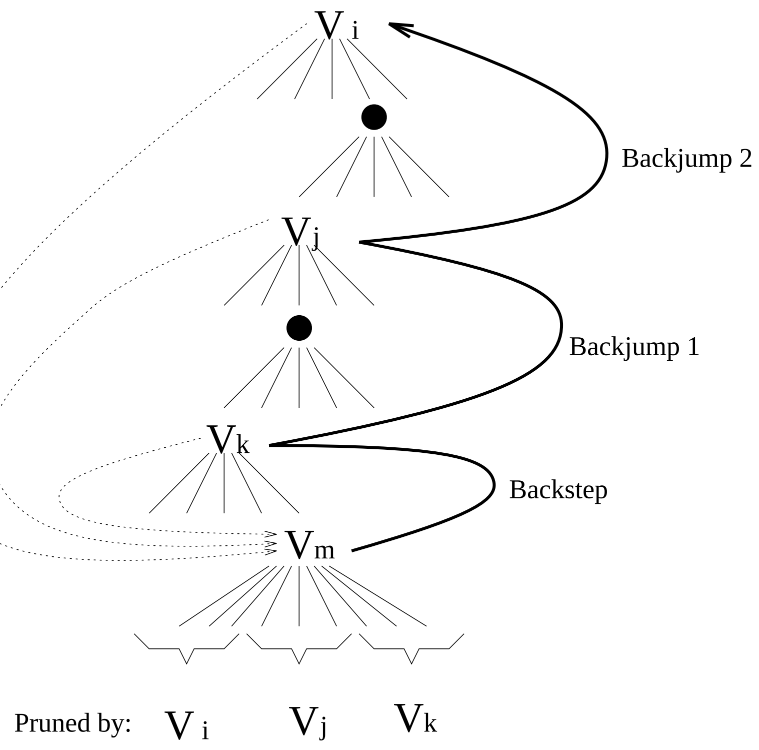


Figure 4.7: Non-uniform domain sizes

thus eliminating large sections of the search tree.

4.4 Conclusions

The major findings reported in this thesis are summarized in this final section.

Variable reordering heuristics can be implemented in BM algorithms contrary to what Prosser claimed in [9]. They can also be included in all of the hybrid algorithms by Prosser, improving them significantly.

Using one extra level of indirection was shown to be sufficient to incorporate variable reordering in any of the discussed algorithms. This extra indirection only has to be used for the current variable and for the part of the algorithm that performs the forward move. Following this basic rule provides a standard way of introducing this heuristic in any of the algorithms.

Backjumping as a backward move in a hybrid non-FC algorithm loses its effectiveness when the variable reordering heuristic is used. Consequently we see no differences in the number of checks performed by Backtracking with variable reordering (BTvar) and Backjumping with variable reordering (BJvar). We can make the same observation concerning the number of checks performed by Backmarking with variable reordering (BMvar) and Backmark Jumping with variable reordering (BMJvar). The equality in the number of consistency checks for these two sets of algorithms indicate that no backjumps occur in the adapted versions. An explanation for this behavior has been given in the discussion of the various algorithms in question.

All the tree-search algorithms perform approximately the same number of consistency checks when variable reordering heuristics are added to them. Even the hybrid algorithms lose the relative increase of performance they received from combining the forward and backward moves of two algorithms. The forward checking hybrids show a substantially larger increase in performance than the tree search algorithms but the value of a more complex backward move also diminishes compared to simple forward checking with variable reordering.

Overall FC-CBJvar turns out to be the best algorithm tested, but FC-BJvar, and especially FCvar, are good and simpler alternatives that only perform significantly worse in very hard problem cases.

The influence of the uniform domain sizes in the problems that were used can be considered a major influence to the loss of effectiveness of the hybrid algorithms. Although the results of the n-Queens problem are always considered to be of less importance because of the particular structure of this problem, the Zebra problem seems to have a more natural structure and is considered to be more representative of real world problems [9].

For future research it might be interesting to look for hybrid algorithms that *do* retain the advantage of combining the forward and backward move of two different algorithms after variable reordering, and that do so even for problems with fixed domain sizes. The incorporation of *value* or *constraint* reordering heuristics, or any combination of them with *variable* reordering, in hybrid algorithms might also produce interesting results. Research into the behaviour of hybrid algorithms with dynamic variable reordering on problems with variable domain sizes is currently underway.

Appendix A

Programming conventions

The program-code in this thesis is presented in a C-like syntax with the following conventions:

- "{" and "}" indicate the begin and the end of a program block;
- A "*" in front of a variable name means that it is a call-by-reference parameter, a variable whose value is returned to the calling function;
- "<>" means not equal;
- All functions return integers;
- $for(i = 1; i <= current; i++)$ represents a *for-loop* with i going from 1 to $current$ in steps of one;
- *if (trivial(current,i)) continue;* causes the next iteration of the enclosing loop to begin if the condition *trivial(current,i)* is true;
- "0" represents False, "> 0" represent True.

Bibliography

[1] Ahrens, W., *Mathematische Unterhaltungen und Spiele (In German)*, B.G. Teubner Publishers, Leipzig 1918-1921

[2] Bitner, J.R. & Reingold, E.M., *Backtrack Programming Techniques*,CACM vol. 18 (1975) 11 651-656

[3] Beek, P. van, *CSP C function library*, University of Alberta, faculty of Computer Science.

[4] Gaschnig, J., *A general backtrack algorithm that eliminates most redundant tests*, Proceedings of the international joint conference on artificial intelligence, Cambridge, MA, 1977, 457

[5] Gaschnig, J., *Performance measurement and analysis of certain search algorithms*, Ph.D thesis CS department, Carnegie-Mellon university PA, 1977

[6] Haralick R.M. & Elliot G.L., *Increasing tree search efficiency for constraint satisfaction problems*, AI vol.14 (1980) 263-313

[7] Kalé L.V., *An almost perfect heuristic for the n nonattacking queens problem*, Information processing letters vol.34 (1990) no.4(apr) 173-178

[8] Knuth, D.E., *Estimating the efficiency of backtrack programs*, Math. Comp. 29 (1975) 121-136

[9] Prosser, P., *Hybrid algorithms for constraint satisfaction problems*, 1993, University of Strathclyde, Scotland, Computational Intelligence, vol.9 (1993), number 3.

[10] Purdom, P.W., *Tree size by partial backtracking*, Siam J. Comput. vol.7 (1978) 4 481-491

[11] Purdom, P.W., & Brown, C.A. & Robertson, E.L., *Backtracking with multi-level dynamic search rearrangement*, Acta Inf. vol. 15 (1981) 99-113

[12] Purdom, P.W. & Brown, C.A., *An average time analysis of backtracking*, Siam j. Comput. vol.10 (1981) 3(aug) 583-593

[13] Purdom, P.W. & Brown, C.A., *An emperical comparison of backtracking algorithms*, IEEE PAMI vol.4 (1982) no.3(may) 309-316

[14] Purdom, P.W., *Search rearrangement backtracking and polynomial average time*, AI vol. 21 (1983) 117-133

[15] Purdom, P.W. & Brown, C.A., *An analysis of backtracking with search rearrangement*, Siam j. Comput. vol.12 (1983) no.4(nov) 717-733

[16] Purdom, P.W. & Brown, C.A., *The analysis of algorithms*, Holt, Rinehart and Winston Inc., 1985

[17] Rossi, F., Petrie, C., Dhar, V., *Equivalence of constraint satisfaction problems*, 1989, Technical report ACT-AI-222-89. MCC Corp., Austin, Texas

[18] Slagle, J.R., *Artificial Intelligence: The heuristic programming approach*, McGraww-Hill Inc., 1971

[19] Stone, H.S. & Sipala, P., *The average complexity of depth first search with backtracking and cutoff*, IBM j. res. develp vol.30 (1986) no.3(may) 242-258

[20] Walker, R.L., *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, 1960

[21] Zabih, R., *Some applications of graph bandwidth to constraint satisfaction problems*, Proc AAAI 1990, vol I, 46-51