Domain filtering can degrade intelligent backtracking search

Patrick Prosser
Department of Computer Science
University of Strathclyde
Glasgow G1 1XH
Scotland

Abstract

This paper presents an improved backjumping algonthm for the constraint satisfaction problem, namely conflict-directed backjumping (CBJ). CBJ is then modified such that it can detect infeasible values and removes them from the domains of variables once and for all. A similar modification is then made to Gaschnig's backjumping routine BJ and to Haralick and Elliott's forward checking routine FC. Empirical analysis shows that these modifications tend to result in an improvement in average performance. The existence of a peculiar phenomenon is then shown: the removal of infeasible values may result in a degradation in the performance of intelligent backjumping algorithms, and conversely the addition of infeasible values may lead to an improvement in performance.

1. Introduction

In the binary constraint satisfaction problem (bcsp) we are given a set of variables and a set of constraints, where each vanable has a discrete and finite domain and each constraint acts between a pair of variables. The problem is to find an assignment of values to variables, from their respective domains, such that the constraints are satisfied [Dechter 1992, Kumar 1992, Mack worth 1992, Meseguer 1989]. This problem may be represented as a graph G, where V(G) is the set of variables and A(G) is the set of constraints. There are a number of tree search algonthms that address this problem, most notably chronological backtracking (BT) [Bitner and Reingold 1975, Golomb and Baumert 1965], backmarking (BM) [Gaschnig 1977 and 1979], backjumping (BJ) [Gaschnig 1979], and forward checking (FC) [Haralick and Elliott 1980]. All these algorithms perform a "depth first" search, and the most primitive of these is BT. When BT instantiates a vanable with a value (the current vanable V_t) it checks backwards against variables that have already been assigned values (the past vanables). If no value can be found for the current variable that is consistent with all of the past vanables, BT steps back to the previous vanable V_{i-1} and attempt a new instantiation for that vanable (and so on). It may be the case that the value assigned to V_{i-1} was not in conflict with V_i, but some other variable higher up in the search tree, V_h, precluded some value from the domain of V,. If the search process could jump back directly to V_h and find a new value for V_h, the search process might then be able to move forward beyond Vi.

Gaschnig's backjumping routine (BJ) attempts to do this. Given the cunent vanable V,, BJ records the "deepest" variable with which V_i checked against. If all values in the domain of V_i failed consistency checks with past vanables, BJ then jumps back to the deepest variable that V_i checked against, namely V_h, and if V_h is re-Instantiated with a new value we may then find a value for V_i. Alternatively, if BJ successfully instantiates the cunent vanable V_i, then the deepest past vanable that V_i checked against will be V_{i-1}. Therefore, if later on in the search process BJ jumps back from V_i to V_h (where $h <$ i), and there are no more values remaining to be tried for V_h, BJ will then "step" back to V_{h-1}. Therefore, we get this mix of "jumping" and "stepping" back. This behaviour can be rectified such that we can continue to jump over constraint violations, and we do this by recording for each vanable V_i the set of vanables that were in conflict with some instantiation of V_i. We call this modified routine conflict-directed backjumping (CBJ). A further modification is then made such that CBJ can detect infeasible values and remove them once and for all. This modified version is called CBJ-DkC (conflict-directed backjumping with directed k-consistency). One would expect that in the worst case CBJ-DkC would perform no worse than CBJ. We show that this is not so. The removal of an infeasible value may result in a reduction in "thrashing" [Mackworth 1977], but this may result in a reduced opportunity for jumping, and the reduction in jumping may outweigh the saving in reduced thrashing.

The algonthms are described in a pseudocode modeled on Pascal and Common Lisp. A fuller description of this language is given in Nadel [1989] and in Prosser [1991]. The following variables are assumed to have been globally declared, n is the number of variables in the constraint satisfaction problem, v is an array of values, such that $v[i]$ $(1 \leq i \leq n)$ is the value assigned to the vanable V_h domain is an array of sequences, such that domain[i] (for $0 \leq i \leq n$) is the domain of the variable V,. Note that domain[0] = nil. current-domain is an anay of sequences, where cunent-domainfi] (for $0 \leq i \leq n$) is the sequence of values in domain[i] that have not yet been shown to be inconsistent with respect to the ongoing search process. current-domain[i] is initialised to be equal to domain[i]. When v[i] is to be instantiated a value is selected from current-domain[i], and if that value is found to be inconsistent with respect to the current search state, then that value is removed from current-domainfi]. When backtracking takes place from v[i] to v[h] (where h < i) cument-domainlj] is reset to domain[j] for all j, where h < j < i. Note that current-domain[0] = nil. C is an n x n array, where C[i j] is the name of a binary predicate (such as <, =, >, ≠, etc) that holds between v[i] and v[j]. If C[ij] = nil then there is no constraint acting between v[i] and v[j]. Therefore we have an extensional representation of constraints (rather than intensional, as a set of compatible pairs). The function check(i,,j) delivers a result of true if C[i,j] = nil, otherwise it delivers the result of applying the relation C[i,j] between the instantiations of v[i] and v[j] (and is counted as a consistency check).

Generally v[i] will be considered to be the current variable, v[h] will be a past variable, and v[j] a future variable (h < 1 < j). It is assumed that all arguments are passed by reference and that the first occurrence of a variable corresponds to an implicit declaration. The algorithms are described in terms of a pair of mutually recursive functions (similar to the style of Dechter and Pearl [1988], and Dechter [1990]). That is, we have a forward move, such as bj-label, and a backward move, such as bj-unlabel. In section 4, the number of "nodes visited" by the search process X is taken to be the number of calls to the forward move x-label. The algorithms address the binary constraint satisfaction search problem [Nudel 1983]. That is, they find the first solution.

2. Conflict-Directed Backjumping (CBJ)

Where BJ steps back from v[h] after jumping back from v[i], the conflict-directed backjurnper (CBJ) continues to jump across conflicts which involve both v[h] and v[i]. CBJ achieves this by recording the set of past variables that failed consistency checks with the current variable (and we refer to this as a "conflict set" as in [Dechter 1990]). If no consistent instantiation can be found for v[i], CBJ then jumps back to the deepest vanable, v[h], that conflicted with v[i]. If on jumping back to v[h] CBJ discovers that there are no more values to be tried in current-domain[h] CBJ then jumps back to v[g], where v[g] is the deepest vanable that was in conflict with either v[i] or v[h].

CBJ maintains a conflict set conf-set[i] for each variable, where the array conf-set is declared globally. Initially each element of conf-set[i] is set to be 10). When a consistency check fails between v[i] and v[h], h is added to the set conf-set[i]. Therefore, conf-set[i] is the subset of the past variables in conflict with v[i]. If there are no remaining values to be tned in current-domain[i], CBJ jumps back to the deepest vanable v[h], where h € conf-set[i] (that is h <-- max-list(conf-set[i]), where the function max-list delivers the largest integer in a set of integers). When jumping back from v[i] to v[h] the information in conf-set[i] is earned upwards to v[h). The array element conf-set[h] becomes conf-set[h] U conf-set[i] - h, the set of variables in conflict with v[h] or v[i]. Therefore, when further backtracking takes place from v[h], CBJ jumps back to v[g], where v[g] is the deepest variable in conflict with either v[h] or v[i].

```
1  PROCEDURE cbj-label (i)
2  BEGIN
3   IF i > n
4   THEN print("solution")
5   ELSE BEGIN
6         consistent ← false;
7         FOR v[i] ← EACH ELEMENT OF current-domain[i]
          WHILE not consistent
8         DO BEGIN
9              consistent ← true;
10             FOR h ← 1 TO i-1 WHILE consistent
11              DO BEGIN
12                  consistent ← check(i,h)
13                  END;
14             IF not consistent
15             THEN BEGIN
16                    current-domain[i]
                          ← remove(v[i],current-domain[i]);
17                    pushnew(h,conf-set[i])
18                    END
19             END
20        IF consistent
21        THEN cbj-label(i+1)
22        ELSE cbj-unlabel(i)
23        END
24 END;
```

In line 17 above, the call pushnew(h,conf-set[i]) adds h to the set conf-set[i] if h is not already a member of conf-set[i]. It is assumed that the loop variable h is available to the statement in line 17, and that h is the value that caused the call to check(i,h) to deliver false.

```
1  PROCEDURE cbj-unlabel (i)
2  BEGIN
3   IF i = 0
4   THEN print("impossible")
5   ELSE BEGIN
6         h ← max-list(conf-set[i]);
7         conf-set[h] ← remove(h,union(conf-set[h],conf-set[i]));
8         FOR j ← h+1 TO i
9         DO BEGIN
10             conf-set[j] ← {0};
11             current-domain[j] ← domain[j]
12             END;
13        current-domain[h] ← remove(v[h],current-domain[h]);
14        IF current-domain[h] ≠ nil;
15        THEN cbj-label(h)
16        ELSE cbj-unlabel(h)
17        END
18 END;
```

CBJ is then realised as cbj-label(l). If we move line 17 in cbj-label to line 12.1 the array element conf-set[i] is updated unconditionally, and CBJ behaves as BJ.

The reasoning behind CBJ might be better understood when viewed from the perspective of de Kleer's ATMS [1986]. We can consider the past vanables as a set of assumptions that are IN (currently believed), and the array element conf-set[i] as a conjunction of assumptions, and therefore an environment. Let S be the set of indices of the past variables ie. S = (1,2,3, ...,i-1), and disallowed[i] = domain[i] - cunent-domain[i], the set of values in domainfi] that have been discovered to be inconsistent with the current search state. We then have the assumed node <disallowed[i],{conf-set[i]},{conf-set[i]}>, where disallowed[i] is the datum, and conf-set[i] is the justification for datum, and consequently the single environment within the label. If $\text{conf-set}[i] \subseteq S$ we then believe that the current search state cannot be extended by any instantiation of v[i] from the set disallowedfi]. Conversely, if conf-setfi] is not subsumed by S we can no longer believe disallowed[i] and we must reset current-domain[i] to be domain[i]. When current-domainfi] is empty (ie. disallowed[i] = domain[i]) we need to force OUT some assumption in conf-set[i], and we choose the most recent assumption, namely max-list(conf-set[i]). This is done implicitly when CBJ backtracks from v[i] to v[h].

CBJ is conservative when it jumps back from v[i] to vfh]. As noted above we can believe disallowed[i] whenever conf-set[i] is subsumed by S. However, when CBJ jumps from v[i], over vfj], to vfh] we automatically reset current-domain[j]. It may be the case that max-list(conf-setfj]) < h and we can continue to believe disallowed[j], and thus prune the search space more efficiently. However, in order to do this we would have to examine all future conflict sets whenever backjumping takes place. CBJ would then look even more like an ATMS. In fact, such an algorithm is described by Rosiers and Bruynooghe [1987] and by Prosser [1989].

CBJ has many features in common with Dechter's graph-based backjumping algorithm GBJ [Dechter 1990]. When GBJ reaches a dead end on v[i] it jumps back to the deepest variable amongst those connected to v[i] in the constraint graph, namely vfh], and if there are no values remaining to be tried for v[h] GBJ jumps back to v[g] where v[g] is the deepest variable connected to either vfi] or vfh]. Therefore we might say that when jumping back BJ is directed by consistency checks that have been performed, CBJ is directed by conflicts, and GBJ is directed by the topology of the constraint graph.

3. Directed Consistency

CBJ can be modified such that it removes values from the domains of variables once and for all, when it can be deduced that these values are infeasible. We may add the following conditional to procedure cbj-unlabel.

```
6.2 IF length(conf-set[i]) = 2
6.3 THEN domain[h] ← remove(v[h],domain[h]);
```

The above modification gives us CBJ-DkC, where DkC stands for "directed k-consistency" [Dechter and Pearl 1988] (and is similar to nth order learning fDechter 1990]). The effect of this modification can be described as follows. Let us assume that CBJ has successfully instantiated v[i-l] and that CBJ moves forwards to v[i]. At that point conf-set[i] = {0}, and current-domainfi] = domainfi]. Further assume that the call to cbj-label(i) fails to find any instantiation of vfi] that is consistent with the past variables. If |conf-set[i]| = 2 then vfi] is in conflict with the instantiation of vfh] and the pseudo variable v[0]. We can then deduce that vfh] is "are-inconsistent" with respect to domainfi]. That is, an arc consistency algorithm [Mackworth 1977, Deville and van Hentenryck 1991] would have removed the value vfh] from domain[h].

Assume that it is not the first time that we have visited vfi], that we successfully instantiate vfi], and that all values in domainfi] are consistent with respect to the past variables. Therefore conf-setfi] = {0}. Assume that we then attempt to instantiate some future variable v[j] and all values in domainfj] conflict with either vfh] or vfi]. We then have conf-setfj] = {0,i,h}, ie. for all values x_j in domainfj], $v[j] \leftarrow x_j$ is inconsistent with vfh] or vfi]. CBJ will then jump back to vfi] and instantiate vfi] with the next value in current-domainfi]. Assume that this process continues until we have exhausted current-domainfi]. We then have conf-setfi] = {O,h}, ie. for all values X_j in domainfj], $v[j] \leftarrow x_j$ is inconsistent with vfh] or $v[i] \leftarrow x_i$, for all values x_i, in domain[i]. Therefore we can remove the value vfh] from domainfh].

We can adopt the same approach with respect to FC and to BJ. In FC we instantiate vfi] with the value k and check forwards against the future variables. Assume that vfi] checks against vfj] and this results in a "domain wipe out" [Nadel 1989] for vfj]. If no other variable checks against vfj] we can then remove k from domainfi] once and for all. This corresponds to "directed arc consistency" (and corresponds to 1st order learning fDechter 1990]), and we can realise this by maintaining a count of the number of variables forward checking against vfj] (as in [Prosser 1991]). We will call this algorithm FC-D2C.

Similarly in backjumping, if no instantiation for v[i] can be found, and all consistency checks from vfi] failed against the single instantiation $v[h] \leftarrow k$, we might then remove k from domainfh]. We can do this by maintaining a flag for each variable, call it instantiated[i], which is initialised to false, and is set to true when bj-label finds an instantiation for vfi] which is consistent with the past variables. If we jump from vfi] to vfh], when length(conf-set[h]) = 2 and instantiatedfi] = false, we can remove the value vfh] from domainfh] (and when we jump from vfi] to vfh] we reset instantiated[j] to false for all $j, h < j \leq i$). This gives us the algorithm BJ-D2C. If we perform the following edits to cbj-label and cbj-unlabel we get BJ-D2C:

(a) In cbj-label move line 17 to line 12.1

(b) In cbj-label replace line 21 with the following segment

```
21   THEN BEGIN
21.1        instantiated[i] ← true;
21.2        cbj-label(i+1)
```

```
21.3    END
```

(c) In cbj-unlabel add the following lines

```
6.1   IF not(instantiated[i]) and length(conf-set[i]) = 2
6.2   THEN domain[h] ← remove(v[h],domain[h]);
10.1              instantiated[j] ← false;
```

More generally, since FC and BJ only reason over failures that occur between pairs of variables we can only detect directed arc inconsistencies (1st order learning). On the other hand, since CBJ reasons over failures within a set of variables, it can detect directed k-inconsistencies (nth order learning).

4. Experimental Evaluation

The following algorithms were compared against each other: BT (naive/chronological backtracking), BJ (Gaschnig's backjumping routine), GBJ (Dechter's graph-based backjumping routine), CBJ (described here), BM (Gaschnig's backmarking routine), FC (Haralick and Elliott's forward checking routine), BJ-D2C, CBJ-DkC, and FC-D2C (again, described here).

The algorithms were applied to 450 instances of the zebra problem, described in [Dechter 1990 and Smith 1992]. That is, 450 different instantiation orders of the zebra were created, and each algorithm was applied to those problems in turn. Table 1 shows the average number of consistency checks performed by an algorithm, the standard deviation, the minimum number of consistency checks performed, and the maximum number performed over the 450 problems. Table 2 shows the same information but with respect to nodes visited.

Algorithm	μ	σ	min	max
BT	6,357,703	15,024,056	8,755	172,074,472
BJ	940,248	2,321,478	1,393	24,393,906
GBJ	1,120,336	2,745,411	1,563	27,475,100
CBJ	132,492	319,107	538	3,991,581
BM	681,802	1,750,022	1,144	18,439,620
FC	66,386	95,855	432	903,400
BJ-D2C	473,437	1,203,653	1,219	18,156,213
CBJ-DkC	61,681	120,009	538	988,049
FC-D2C	47,492	74,414	396	885,786

Table 1. Consistency Checks

Algorithm	μ	σ	min	max
BT	1,249,087	2,845,631	1,893	29,942,330
BJ	173,620	397,502	274	4,056,985
GBJ	207,848	481,350	364	4,413,676)
CBJ	24,178	55,954	111	632,847
BM	1,249,087	2,845,631	1,893	29,942,330
FC	7,092	9,922	33	76,405
BJ-D2C	87,858	215,807	229	3,077,572
CBJ-DkC	11,317	21,790	111	205,774
FC-D2C	5,422	7,793	33	75,541

Table 2. Nodes Visited

If we take consistency checks performed as a measure of search effort we may rank the algorithms as follows: FC-D2C, CBJ-DkC, FC, CBJ, BJ-D2C, BM, BJ, GBJ, BT. With respect to nodes visited the algorithms are ranked: FC-D2C, FC, CBJ-DkC, CBJ, BJ-D2C, BJ, GBJ, (BM and BT).

The algorithms were then applied again to 100 instances of the zebra problem, and the cpu time was measured. Table 3 below shows the average cpu time used (on a SPARCstation IPC, with 24 mega-bytes of memory, using Sun Common Lisp 4.0) by the algorithms for solving an instance of the problem, and the average number of consistency checks performed in a second.

Algorithm	seconds	checks/sec
BT	123	12,221
BJ	32	8,771
GBJ	29	10,311
CBJ	6	8,953
BM	102	1,659
FC	5	7,707
BJ-D2C	13	7,682
CBJ-DkC	3	8,769
FC-D2C	3	7,588

Table 3. CPU Time

Although BT performed on average 8 times as many consistency checks as BM (Table 1) BT took only 20% longer to run than BM (Table 3). This is due to the poor "checking rate" of BM (and this is explained more fully in [Prosser 1991 and 1993]) CBJ has a higher checking rate than BJ. Therefore, not only does CBJ perform less checks than BJ, it performs these checks with less overheads (these tests used the more efficient version of BJ described in [Prosser 1991], rather than the derived version here). This is because CBJ updates conf-set[i] conditionally, and BJ updates max-check[i] unconditionally. Generally, there is an insignificant overhead associated with the modifications performed to BJ (to give us BJ-D2C), CBJ (giving CBJ-DkC, and FC (to FC-D2C). These modifications resulted in a reduction in consistency checks performed, nodes visited, and a reduction in run time. Therefore, with respect to run time the algorithms may be ranked: FC-D2C, CBJ-DkC, FC, CBJ, BJ-D2C, BJ, BM. With the exception of BM, this ranking agrees with those above, and in fact there is little to choose between CBJ-DkC and FC-D2C.

5. The Bridge (and the Long Jump)

It was expected that CBJ-DkC would always perform at least as well as CBJ. However, on analysing the experimental results it was discovered that out of the 450 problem instances there were 2 cases where CBJ performed better than CBJ-DkC. This was a surprise. One of these problems was then examined in detail. This was the problem with the instantiation order: <Water, Tea, Coffee, Japanese, Kools, Blue, Ukranian, Chesterfield, Old-Gold, Zebra, Horse, Fox, Orange-juice, Yellow, Snails, Red,

Green, Englishman, Lucky, Dog, Spaniard, Parliament, Ivory, Norwegian, Milk>. During the search process CBJ-DkC discovers, amongst other infeasibilities, that there is no solution to the problem when Spaniard is assigned the value 1 (ie. $v[21] \leftarrow 1$). Therefore, the value 1 is removed from domain[21]. In some latter stage in the search process v[21] again becomes the current variable and CBJ-DkC considers the instantiation $v[21] \leftarrow 2$. At the same point in the search space CBJ considers the instantiation $v[21] \leftarrow 1$. The two search trees now differ significantly, and in CBJ's search tree it is possible to jump back to a conflicting variable higher up in the search tree than CBJ-DkC.

More generally, CBJ-DkC may remove an infeasible value k from the domain of a variable v[i]. At some later stage in the search process CBJ may move forwards from v[i-l] to v[i], and be unable to re-instantiate v[i] with the value k. CBJ-DkC may then jump back to v[h]. At the same point in the search tree CBJ is allowed to make the instantiation $v[i] \leftarrow k$, and move forwards to v|j]. CBJ may then jump back from v[j] to v[g], where g < h. Therefore, the value k has acted as a bridge that allows the search process to move from one area of the search space to another, where it can then make a "long jump" back to a conflicting variable.

To confirm this analysis, the value 1 was removed from domain[21], the problem was reset, and CBJ and CBJ-DkC were re-run. It was expected that CBJ would be unable to "cross the bridge" and unable to make "a long jump". With the bridge in place CBJ performed 10,746 checks, and visited 1,974 nodes (and CBJ-DkC performed 13,097 checks, and visited 2,390 nodes). With the bridge removed CBJ performed 13,798 checks, and visited 2,532 nodes (CBJ-DkC performed 13,029 checks and visited 2,385 nodes). This implies that the removal of an infeasible value from the domain of a variable may result in a degradation in the performance of an algorithm that jumps back to the cause of a conflict (such as BJ, CBJ, or any of the hybrid derived from these algorithms [Prosser 1991 and 1993]).

6. Conclusion

A new algorithm has been presented, CBJ. It has been shown that BJ can be derived from CBJ, and that BJ might be considered to be a degenerate form of CBJ. CBJ was then modified (the addition of a single conditional expression) such that infeasible values can be detected and removed once and for all. A similar technique was applied to backjumping and forward checking. Empirical evidence suggests that these modifications result in an improvement in the performance of these algorithms *on average*

The removal of infeasible values has revealed a disturbing phenomenon, namely that this can lead to a degradation in the performance of a "conflict directed" backjumping algorithm. It has (almost) become an article of faith that if we remove infeasible values [Mackworth 1977, Deville and van Hentenryck 1991, Freuder 1982] or redundant values [Benson and Freuder 1992, Freuder 1991] from the domains of variables, the subsequent search algonthm will be presented with an easier task. We have been lead to believe this because "the subsequent search algonthm" is generally assumed to be a chronological backtracker (BT, BM, FC), or the effect of removing a bndge has been masked by a reduction in thrashing. We should now assume that increased consistency, or the removal of redundancies, can only guarantee a reduction in search effort if that search is unintelligent (such as a chronological backtracker). Conversely, we should expect that we can improve the performance of an intelligent backjumping algorithm by adding an infeasible value to the domain of a variable.

References

[Benson and Freuder 1992] B.W. Benson and E.C. Freuder, Interchangeability preprocessing can improve forward checking search. *Proceedings ECAI-92* 28-30 (1992)

[Bitner and Reingold 1975] J.R. Bitner and E. Remgold, Backtrack programming techniques, *Commun. ACM,* 18 (1975)651-656

[Dechter and Pearl 1988] R. Dechter and J. Pearl, Network-based heuristics for constraint-satisfaction problems, *Artif. Intell.* 34(1) (1988) 1-38

[Dechter 1990] R. Dechter, Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition, *Artif. Intell.* 41 (3) (1990) 273-312

[Dechter 1992] R. Dechter, Constraint Networks, In: *Encyclopedia of Artificial Intelligence, Second Edition,* Volume 1, 276-285, Wiley-Interscience Publication, editor-in-chief S.C. Shapiro, 1992.

[de Kleer 1986] J. de Kleer, An assumption-based TMS *Artif Intell.* 28(1986)127-162

[Deville and van Hentenryck 1991] Y. Deville and P. van Hentenryck, An efficient arc consistency algorithm for a class of csp problems, *Proceedings IJCA1-91,* (1991) 325-330

[Freuder 1982] E.C. Freuder, A Sufficient Condition of Backtrack-Free Search, *J. ACM* 29 (1) (1982) 24-32

[Freuder 1991] E.C. Freuder, Eliminating interchangeable values in constraint satisfaction problems, *Proceedings AAAJ-91* (1991) 227-233

[Gaschnig 1977] J. Gaschnig, A General Backtracking Algonthm that Eliminates Most Redundant Tests, *Proceeding IJCA1-77* (1977) 457

[Gaschnig 1979] J. Gaschnig, Performance measurement and analysis of certain search algonthms, Tech. Rept. CMU-CS-79-124, Carnegie-Mellon University, Pittsburgh, PA (1979)

[Golomb and Baumert 1965] S.W. Golomb and L.D. Baumert, Backtrack Programming, */. ACM,* 12 (1965) 516-524

[Haralick and Elliott 1980] R.M. Haralick and G.L. Elliott, Increasing Tree Search Efficiency for Constraint Satisfaction Problems, Artif. Intell 14 (1980) 263-313

[Kumar 1992] V. Kumar, Algorithms for constraint satisfaction problems: a survey, AI magazine 13 (1) (1992) 32-44

[Mackworth 1977] A.K. Mackworth, Consistency in Networks of Relations, Artif. Intell 8 (1) (1977) 99-118

[Mackworth 1992] A.K. Mackworth, Constraint Satisfaction, In: Encyclopedia of Artificial Intelligence, Second Edition, Volume 1, 285-293

[Meseguer 1989] P. Meseguer, Constraint satisfaction problems: an overview, AICOM 2 (1) (1989) 3-17

[Nadel 1989] B.A. Nadel, Constraint Satisfaction Algorithms, Computational Intelligence 5(4): 188-224, 1989

[Nudel 1983] B.A. Nudel, Consistent Labelling Problems and their Algorithms: Expected Complexities and Theory Based Heuristics, Artif Intell. 21 (1983) 135-178

[Prosser 1989] P. Prosser, A reactive scheduling agent, Proceedings UCA1-89 (1989) 1004-1009

[Prosser 1991] P. Prosser, Hybrid algorithms for the constraint satisfaction problem, Tech. Rept. AISL-46-91, Department of Computer Science, University of Strathclyde, Glasgow, Scotland (1991). To appear in Computational Intelligence 9(3), August 1993

[Prosser 1993] P. Prosser, BM+BJ=BMJ, Proceedings CA1A-93, (1993)257-262

[Rosiers and Bruynooghe 1987] W. Rosiers and M Bruynooghe, Empirical study of some constraint satisfaction algorithms, in Artificial Intelligence II: Methodology, Systems, Applications Edited by P. Jorrand and V. Sgurev, Elsivier Science Publishers B.V. (North-Holland) 1987

[Smith 1992] B.M. Smith, How to solve the zebra problem, or path consistency the easy way, Proceedings ECAI-92 (1992) 36-37