Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems

Pieter Andreas Geelen
Vrije Universiteit Amsterdam, Dept. of Computer Science, Artificial Intelligence Group
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

Abstract. Many AI problems can be formulated as *Constraint Satisfaction Problems (CSP)*. In this paper, we present problem-independent heuristics for evaluating decisions with a clear combinatorial interpretation. As a side-effect, they support constraint-preprocessing (arc-consistency). Next, we introduce a "dual-viewpoint" approach for a special (broad) class of CSP. This approach allows suitable extensions to many heuristics, and in particular of the heuristics presented in this paper. Test results on N-queen problems validate the usefulness of the concepts presented in this paper.

1. Introduction

Many AI problems can be formulated as *Constraint Satisfaction Problems (CSP,* see [14] for a good overview). A CSP is represented by a set of n variables, for each variable X_i a finite domain D_i (with cardinality a or less), and a set of e *constraints*. A constraint R_j denotes the values mutually compatible for variables $X_{j1},...,X_{jP_j}$.
P_j is called the *arity* of R_j.
Given a set S of assignments of certain values to certain variables, we define CONSISTENT(S) to be true iff these assignments do not violate any constraints. A *solution* to the CSP will now be any set

$$\mathbf{SOL} = \{X_1=V_1, X_2=V_2, ..., X_n=V_n\} \quad (V_i \in D_i)$$

for which CONSISTENT(**SOL**). We can restrict ourselves to *binary* CSP (in which all constraints have an arity ≤ 2) without loss of generality [15].

2. "BackTrack" and Forward Checking

We will concentrate on the problem of finding *one* solution to a CSP. As a basis for our heuristics, we will use the *BackTrack* algorithm ([8],[5],[2]): pick a variable that has not yet been assigned a value, pick a value for it, and assign it to the variable. If a constraint-violation is then noticed, go back ("backtrack") and pick an alternative value. If no alternatives exists, backtrack even further. Continue until a solution is found or until all possible sets of assignments have been tried (note that there are $O(a^n)$ such sets!). We furthermore assure *node consistency* beforehand [11] by removing from D_i every V_i for which $\neg$CONSISTENT($\{X_i=V_i\}$). Finally, whenever an assignment $X_i=V_i$ is made, we re-assure node consistency by removing all $V_j \in D_j$ for which $\neg$CONSISTENT($\{X_j=V_j, X_i=V_i\}$) (i.e. we perform *Forward Checking* [9]). For *binary* CSP, checking wether two assignments are consistent with each other takes O(1) time, so forward checking takes $O(an)$ time per assignment.

Of course, we would like BackTrack to select values that do not cause constraint-violations. Such "perfect" value-selection is usually impossible, in which case the sequence in which *variables* are selected is often very important, too. First of all, the success of the value-selection heuristics may *depend* on which variable is selected. Furthermore, variable selection often determines how soon a *constraint-violation* is noticed, and thus how much time is wasted working on a "doomed" attempt (assuming ordinary backtracking. See [4],[6],[7] on alternative ways to handle constraint-violations).
In this paper, we will concentrate on heuristics for variable and value-selection specifically suited to find *one* solution to a CSP. To demonstrate our ideas, we will use the N-queen problem (the problem of placing N queens on an N*N chessboard in such a way that they do not attack each other. A problem that is purely academic: [16] shows a linear-time algorithm for generating a solution for any N>3. See [1] for a refinement yielding more solutions.) This problem can be formulated as a CSP with N variables, each with N values in its domain (i.e. $a=n=N$). The assignment $X_i=k$ represents a queen placed at row i, column k. The constraints are (for $i \neq j$):

$X_i \neq X_j$ (no two queens in the same column)
$|X_i-X_j| \neq |i-j|$ (no two queens on the same diagonal)

Given a set of assignments S to have been effected, and X_i a *future* variable (one that is as yet unassigned), we define $DOM_S(X_i)$ as the set of "useful" values for X_i (values that, given S, can be assigned to X_i such that no constraint is violated), i.e.

$$DOM_S(X_i) = \{V_i | V_i \in D_i \wedge \text{CONSISTENT}(S \cup \{X_i=V_i\})\}$$

The Forward Checking algorithm keeps track of $DOM_S(X_i)$ for every future variable X_i. Given some S, we next define the *number* of values that will *still* be useful for a future variable X_j after assigning $V_i \in DOM_S(X_i)$ to X_i ($i \neq j$), as

$$LEFT_S(X_j | X_i=V_i) = |DOM_{S \cup \{X_i=V_i\}}(X_j)|$$

Conversely, we define the number of useful values for X_j that will then *not* be useful any more as

$$LOST_S(X_j | X_i=V_i) = |DOM_S(X_j)| - LEFT_S(X_j | X_i=V_i)$$

For binary CSP, we can easily proof that :

$$LEFT_S(X_j | X_i=V_i) = |\{V_j | V_j \in DOM_S(X_j) \wedge \text{CONSISTENT}(\{X_i=V_i, X_j=V_j\})\}|$$

Thus, determining a LEFT (or LOST) takes O(a) time.

ECAI 92. *10th European Conference on Artificial Intelligence* Edited by B. Neumann
Published in 1992 by John Wiley & Sons, Ltd

3.1. Value selection

In order to find *one* solution to a CSP, it seems a good heuristic to select the *least constraining value* for a variable, i.e. the value "least reducing the possibility to assign values to the *other* variables". Since assignments may reduce (and will certainly never increase) the number of useful values for other variables, and thus reduce the chance that these variables are left with at least *one* useful value, the concept of "least constraining" may well be defined in terms of LEFT or LOST. A well-known formula for determining how "constraining" a value $V_i \in DOM_S(X_i)$ is for a future variable X_i, comes down to

$$\text{total-cost}_S(X_i=V_i) = \sum_{j\neq i, X_j \text{ a future variable}} \text{LOST}_S(X_j \mid X_i=V_i) \quad (1)$$

It represents the number of individual assignments that we will not be able to make any more after assigning V_i to X_i (e.g. for the N-queen problem, it represents the total number of "free squares" that are lost to other rows if a queen is placed at row i, column V_i). Keng&Yun ([10]) present a similar function. $\text{LOST}_S(X_j|X_i=V_i)$ divided by $|DOM_S(X_j)|$ results in the *percentage* of useful values that are lost to X_j. Slightly paraphrasing their arguments, they consider this percentage an indication of the "dislike" of X_j to the assignment $X_i=V_i$. To see how much an assignment is disliked "in general", they summarize the percentages:

$$\text{cruciality}_S(X_i=V_i) = \sum_{j\neq i, X_j \text{ a future variable}} \frac{\text{LOST}_S(X_j|X_i=V_i)}{|DOM_S(X_j)|} \quad (2)$$

Now, formulae (1) and (2) both reason from the point of view of what an assignment *costs*. But to us it seems far more important that enough useful values will be *left* afterwards. Furthermore, summation suffers from the disadvantage that there is no difference between 0+6 and 3+3, for instance, even though the distinction is quite important. Suppose we have variables X_1,X_2 and X_3, each having 6 useful values. Suppose a certain assignment to X_3 would leave X_1 with 6, and X_2 with 0 useful values (a doomed situation), while another would leave them both with 3 useful values. Formulae (1) and (2) evaluate such assignments as "equally good". In fact, between 6+0 and 2+3, the doomed 6+0 is preferred. We propose maximizing

$$\text{promise}_S(X_i=V_i) = \prod_{j\neq i, X_j \text{ a future variable}} \text{LEFT}_S(X_j|X_i=V_i) \quad (3)$$

in order to select a value V_i for variable X_i, which (like formula (1), and contrary to formula (2)) has a clear interpretation : it represents the number of different *sets* of assignments that, after assigning $X_i=V_i$, can still be made such that no constraint *on* X_i is violated. It thus represents an upperbound on the number of different solutions **SOL** such that $S \cup \{X_i=V_i\} \subseteq \mathbf{SOL}$. Thus the first assignment in the example above leaves *at most* 0*6=0 possible solutions, while the second one leaves *at most* 3*3=9 solutions. The idea underlying the maximization of formula (3) is that, *assuming every set of assignments has an equal chance of being a solution,* we have the best chance of finding *one* solution if we try to leave as many chances as possible. As with summation, the product of large values is larger than the product of small values. But note that multiplication differentiates far better than summation: since $(a+b)(a-b) = a^2-b^2$, the more two numbers differ from their average, the less their product will be. Thus, 0*6 < 1*5 < 2*4 < 3*3. Also note that the promise is 0 (minimal) if and only if the situation is doomed, i.e. some variable would be left without useful values. Using summation formulae such as (1) or (2), the best we may hope for in such cases is that the results get high enough.

3.2. Variable Selection

A well-known variable-selection heuristic is to select the *most constrained* variable (e.g. the variable with the fewest useful values [14]). In [10], Keng&Yun explicitly use the evaluations of the *values* of a variable in order to determine how constrained the variable itself is. The idea can be demonstrated as follows: suppose that there are no constraints on a variable A (i.e. any value can be assigned to A "without cost"), while every useful value for a variable B heavily constrains all other future variables. Then the number of useful values is not enough indication of how constrained a variable is: it depends far more on how constraining the possible *values* for the variable are. The more constrained the useful values of a variable are, the less we can afford to postpone choosing one. Suppose we determine (Figure 1) , for every square of a 4-Queen problem, the number of squares in other rows that are attacked by it (i.e. using formula (1)).

X_1	6	6	6	6
X_2	6	8	8	6
X_3	6	8	8	6
X_4	6	6	6	6

Figure 1. The number of squares attacked in other rows.

Now, X_2 and X_3 can be considered more constraining because their *values* are more constraining. In this case it is easily seen, because *every* value of X_2 and X_3 costs at least as much as *any* value of X_1 and X_4. The point, of course, is how we can combine the evaluations of useful values into an evaluation of a variable itself *in general.* In [10] a formula is maximized that is equivalent with:

$$\text{criticality}_S(X_i) = \prod_{V_i \in DOM_S(X_i)} \frac{1}{(1+|DOM_S(X_i)|*\text{cruciality}_S(X_i=V_i))} \quad (4)$$

Why the crucialities are combined in exactly *this* way is not quite clear. Nor does the result of this formula have any clear interpretation (except that the lower the value, the better). We minimize the following formula:

$$\text{promise}_S(X_i) = \sum_{V_i \in DOM_S(X_i)} \text{promise}_S(X_i=V_i) \quad (5)$$

The result of this formula is the *total* number of value-sets that can be assigned to *all* future variables (including X_i) such that no constraint *on* X_i is violated, and thus represents an upperbound for the number of different solutions **SOL** to the CSP such that $S \subseteq SOL$.
Consider the promise of the assignments (squares) for the initial 4-queen problem (Figure 2), and the promise of each variable (row). X_2 provides an upperbound of 20 solutions, while X_1 promises up to 28 solutions. The constraints on X_2 appearantly constrain the other variables more (at least *more directly, more noticable)* than do those on X_1, leading to a more constrained - and more realistic - result for X_2.

X_1	8	6	6	8	28
X_2	8	2	2	8	20
X_3	8	2	2	8	20
X_4	8	6	6	8	28

Figure 2

X_1			2	1	3
X_2					
X_3			0	1	1
X_4		2		0	2

Figure 3

Given the results as shown in Figure 2, we could best assign either 1 or 4 to either X_2 or X_3. Picking X_2=1, we get the situation of Figure 3. It promises at most one complete solution (X_3's promise). Since X_3 is the least promising variable, and "4" is its most promising value, X_3=4 is effected. X_1=3 and X_4=2 follow immediately.
Given the set **S** of variables which have already been assigned a value, $promise_S(X_i)$ actually represents the number of solutions to a simplified CSP, in which only the future variables, their useful values, and constraints between X_i and other future variables are considered. In the same way, $promise_S(X_i=V_i)$ represents the number of solutions to an even simpler (more restricted) CSP (one where the unary constraint "$X_i=V_i$" is added). So the heuristic underlying formula (3) could be formulated as "an assignment that has more solutions in a simplified CSP *promises* more solutions in the current, difficult CSP". It could therefore be seen as a variation of the techniques presented in [5] (our simplification being more drastic but allowing cheap *calculation* of the number of solutions).

4. Algorithms using the formulae

The simplest algorithm we will consider is LD ("least-domain"). It selects the variable with the fewest useful values, and uses a formula (such as (1), (2), or (3)) to select its "least constraining" useful value. For LD algorithms, every useful value for the selected variable must be tested against all other future-variable/useful-value-pairs, and will thus take $O(na)$ time, so making one assignment will take $O(na^2)$ time[1].

[1] Of course, problem-*dependent* optimizations might be made. For the N-queen problem, for instance, no square can attack more than three squares in any other column j. This allows us to reduce time complexities by a factor **a**.

An algorithm which evaluates *all* useful values for *all* future variables in order to make selections will be called FE (for "full evaluation") and will take $O(n^2a^2)$ time per assignment. For both LD and FE, space complexity is $O(na)$, since we must be able to mark every value for every variable "useful" or "useless";
We extended implementations with the "domino effect": when a variable is left with exactly *one* useful value, we immediately assign it (skipping the calculations).

4.1. Value pruning and arc consistency

As a side effect of "Full Evaluation", we can constantly assure *arc-consistency* ([11][12]). If $LEFT_S(X_j|X_i=V_i)=0$, then the assignment $X_i=V_i$ would leave X_j without useful values. We could therefore immediately prune (=mark as useless) V_i for X_i. Of course, LEFTs calculated *before* this pruning may get out to date. For instance: if we had pruned the zero-promise assignments X_3=3 and X_4=4 in Figure 3, and then *recalculated* all promises, X_1=4 would also have become zero-promise, and the solution would have been obvious. But recalculation takes $O(n^2a^2)$ time, and we might have to do it **na** times! Pruning some unpromising values may not be worth this.
However, note that if a value V_i is pruned for X_i, *only* X_i is left with less useful values than before. Now, $CONSISTENT(S \cup \{X_i=V_i, X_j=V_j\})$ means that V_i would be a useful value for X_i after assigning $X_j=V_j$. So, pruning V_i, we only have to decrement $LEFT_S(X_i|X_j=V_j)$ by 1 for all $X_j=V_j$ for which $CONSISTENT(S \cup \{X_i=V_i, X_j=V_j\})$. Should this cause $LEFT_S(X_i|X_j=V_j)$ to become 0, we prune V_j for X_j in the same way. If, during the pruning process, any variable is left without *any* useful values, we can break off and backtrack immediately. Recalculations needed when pruning a value thus need only $O(na)$ time. Since there can't be more than **an** useful values of future variables, FP ("full pruning") takes $O(n^2a^2)$ time, just like FE. The algorithm described here is in fact similar to the "optimal complexity arc-consistent algorithm AC-4" presented in [13], generalized to an algorithm for K-consistency in [3]. However, if we perform similar decrementations for the values pruned by the Forward Checking algorithm, we *never* have to recalculate all LEFTs except when backtracking. In fact, if a solution is found without backtracking, we calculate the LEFTs exactly *once*, in $O(n^2a^2)$ time; we prune less than **na** assignments, updating LEFTs at $O(na)$ time per pruning; we calculate promises **n** times at $O(n^2a)$ time each. Best case time-complexity for FP algorithms is thus $O(n^2a^2+n^3a)$, or $O(n^4)$ for N-queen problems. The same can be done for the FE-algorithms. (But note that storing LEFTs introduces a $O(n^2a)$ space complexity).

5. Permutation Problems

In this last section we will show that many heuristics, and formulae 3 and 5 in particular, can be extended to solving *Permutation Problems (PP)*. These are CSP for which **n=a**, and for which $X_i \neq X_j$ for any $i \neq j$. So for PP,

every value must be assigned to exactly *one* variable. What is interesting about PP is that one could consider them from an "inverted viewpoint", where the problem is to "find a variable for every value". Everything that can be calculated, deduced or estimated from this inverted viewpoint also has validity for the "normal" viewpoint.
In fact, at any moment, in any PP, we can

1 - choose which "viewpoint" to take, and/or
2 - combine the results of our calculations, deductions or estimates of both viewpoints.

Of course, we would need heuristics, both for deciding when to take which viewpoint, and for combining the results. A heuristic for choosing a viewpoint is easy: at every step of BackTrack, the number of future variables equals the number of "future values", i.e. both "versions" of the PP are always in a comparable situation. Given *any* heuristic for determining the most constrained variable: pick the most constrained variable in the normal viewpoint, except if the "most constrained value" in the inverted viewpoint is even *more* constrained. Consider the LD-variations, for example. In Figure 3, all future variables have 2 useful values left. From the inverted viewpoint, however, value 2 only has *one* "useful variable" left. A "dual-viewpoint" LD-variation would thus concentrate on picking a *variable* for *value* 2. The dual viewpoint also strengthens the "domino"-effect and the "pruning-power" of FP-algorithms (examples of combining the *deductions* of both viewpoints). But for formulae 3 and 5, we also have a clear way to combine the *calculations* of both viewpoints. Consider the definitions of the "inverted LEFT" and "inverted promises":

$$LEFT^{inv}{}_S(\ V_j\ |X_i{=}V_i\ ) = \\ |\{X_j|\ V_j{\in}DOM_S(X_j) \wedge CONSISTENT(\{X_i{=}V_i,X_j{=}V_j\})\}|$$

$$Promise^{inv}{}_S(V_i{=}X_i) = \\ \prod_{j\neq i,V_j \text{ a future value}} LEFT^{inv}{}_S(\ V_j\ |\ V_i{=}X_i\ )$$

$Promise^{inv}{}_S(X_i{=}V_i)$ represents the number of possibilities left to "assign (other) variables to the other values" such that no constraints on V_i are violated, after we assign $X_i{=}V_i$. Since a promise represents an upper-bound for the number of solutions, we can combine the promises of both viewpoints as follows:

$$CPromise(X_i{=}V_i){=}CPromise^{inv}(X_i{=}V_i) = \\ \mathbf{min}(\ Promise(X_i{=}V_i),Promise^{inv}(X_i{=}V_i)\ )$$

and evaluate variables and values using

$$CPromise(X_i) = \sum_k CPromise(X_i{=}V_i)$$
$$CPromise^{inv}(V_i) = \sum_i CPromise^{inv}(X_i{=}V_i)$$

For the "dual-viewpoint" algorithm, the evaluations for empty N*N chessboards will now be truly symmetrical (e.g. in Figure 2, the promises for $X_2{=}1$, $X_3{=}1$, $X_2{=}4$ and $X_3{=}4$ would also be 6).

5.1. Partial Permutation Problems

If we drop the requirement **a=n** (i.e. allow **a>n**) we get the class of *Partial Permutation Problems (PPP)*, e.g. resource scheduling problems where resources have a capacity of one. We lack the space to do more than give an indication of our work. We present two approaches to handle PPP. The first is to transform the PPP into a PP by adding "fake" variables $X_{n+1},..,X_a$, on which no constraints hold except that globally, $X_i{\neq}X_j$ for any $i{\neq}j$. At first sight this may seem an extreme measure, but note that all fake variables start out with the same useful values, and that values pruned for a fake variable can be pruned for *all* fake variables. So they always have the same useful values, LEFTs, and promises: we only need to keep track of one of them, and remember there are **a-n** of them. Also note that in the *normal* viewpoint, the *real* variables will always be preferred, since they are always more constrained. In the *inverted* viewpoint, however, the most constrained values are those of the fake variables: the effect of inverted reasoning is that the **a-n** most constraining values are "removed" from the domains of the real variables. Mixing viewpoints, we get the best of both worlds.
An interesting alternative is to extend formula (3) so it can handle inverted PPP. Figure 4a shows the inverted viewpoint on the problem of placing 3 queens on a 3x4 chessboard. In each square, we show the LEFTs for every other "future value", should a queen be placed in that square.

X_1	1,1,2	1,1,1	1,1,1	1,1,2
X_2	0,2,2	0,0,2	0,0,2	0,2,2
X_3	1,1,2	1,1,1	1,1,1	1,1,2

Figure 4a

X_1	5(4)	3(2)	3(2)	5(4)
X_2	4(4)	0(1)	0(1)	4(4)
X_3	5(4)	3(2)	3(2)	5(4)

Figure 4b.

Formula 3 would take the product of those LEFTs, but this time, the problem is to assign only *two* values to *two* other variables. To see in how many ways "p out of q values" can be assigned to p variables, we need, *for every subset of p values,* the product of their LEFTs. The sum of all these products is the upperbound on the number of solutions we want. So, in Figure 4a, for $X_1{=}1$, the number of ways to place the other *two* queens would be 1*1+1*2+1*2 = 5.
In general, there are q-over-p sets, but we managed an O(p(q-p))-algorithm for the calculations, of which we show the formal version without explanation and with only an indication of the proof:

```
T[0]=1; T[1]=T[2]=...=T[p]=0;
for j := 0 to q-p do
    for i := 1 to p do T[i]= T[i] + T[i-1] * E_{i+j}
```

Summarizes in T[p] the products of every subset of p elements out of the set $\{E_1,...E_q\}$. Proof is by induction on j and i, showing that for every j the algorithm summarizes in T[i] the products of every subset of i elements out of the set $\{E_1,...,E_{j+i}\}$. For p=q, it just implements formula 3.

Figure 4b shows the results of the combinatorial calculations, and between brackets the promises calculated in the "normal" viewpoint (using the ordinary formula

(3)). The minimum of these two would be the most realistic estimate.

6. Test Results and conclusions

The following table shows runtest results for 100 N-queen problems (4≤N≤103) for several algorithms:

Algorithm	average number of backtracks	nr of back-track-free solutions	maximum number of backtracks
LD +formula 1	>45000	20	>2500000
LD +formula 2	>33000	15	>2500000
LD +formula 3	1205	3	92379
LD +formula 1, dual	21.6	26	548
FE +formulae 2&4	9.5	71	812
FE +formulae 3&5	5.1	68	266
FP +formulae 2&4	5.6	81	497
FP +formulae 3&5	4.2	68	224
FE +formulae 3&5, dual	0.38	90	12

Note that the dual-viewpoint approach shows dramatic improvement even for LD1. The dual-viewpoint version of FE35 (using the CPromise-formulae discussed in section 5) generated backtrack-free solutions for almost all problems tested, and outperforms even the full-pruning FP24/FP35 algorithms. We did not implement ^{2}FP, since ^{2}FE35 didn't leave much opportunity for improvement anyway. As Nudel [15] notes, less backtracking is interesting only if it means lower run-time. The number of constraint-checks made by different algorithms for different N are plotted in Figure 5. Time spent on backtracking is not visible in this figure (the run-time plot is similar to Figure 5, but with higher "peaks").

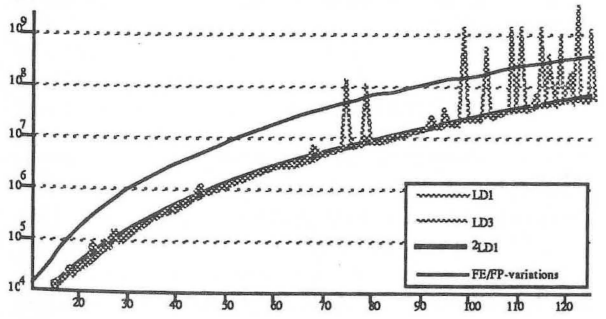

Figure 5. Nr. of constraint checks for algorithms

We have presented two heuristic formulae which provide upperbounds for the number of solutions to a binary CSP. Efficient algorithms can be constructed using these formulae. We introduced a "dual-viewpoint" approach for a special (broad) class of CSP, which provides more information to work with for many heuristics, and for own formulae in particular. Test results indicate the usefulness of both formulae and approach. We are currently looking into other, less evenly constrained CSP on which to test the formulae and ideas presented in this paper.

Acknowledgements.

The research described here is supported partly by SKBS and partly by NWO. The author is much indebted to Guszti Eiben, Wojciech Kowalczyk, Zsofia Ruttkay, Jan Treur and Arnold Wentholt of the Vrije Universiteit Amsterdam for their critical proofreading and valuable suggestions.

References

[1] B. Abramson, M. Yung. Divide and Conquer under Global Constraints: A Solution to the N-queens Problem, *Journal of Parallel and Distributed Computing, 6* (1989), pp.649-662

[2] C.A. Brown, P.W.Jr Purdom. How to search efficiently, *Proc. IJCAI* (1981) pp.588-594

[3] M.C. Cooper. An Optimal K-Consistency Algorithm, *Artificial Intelligence* **41** (1989) pp.89-95.

[4] R. Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning and Cutset Decomposition, *Artificial Intelligence* **41** (1990) pp.273-312.

[5] R. Dechter, J. Pearl. Network-based Heuristics for Constraint Satisfaction Problems, *Artificial Intelligence* **34** (1988), pp.1-38.

[6] J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems, *Proceedings 2nd National Conference Canadian Society for Computational Studies of Intelligence*, Toronto, Ontario, 1978.

[7] J. Gaschnig, Performance measurement and analysis of certain search algorithms, *Ph.D. Thesis,* Dept. Computer Science, Carnegie-Mellon University, 1979.

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

[9] R.M. Haralick, G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence* **14** (1980), pp.263-313.

[10] Keng, N. and Yun, D.Y.Y, A Planning/Scheduling Methodology for the Constrained Resources Problem, *Proc. IJCAI* (1989), pp.999-1003.

[11] Mackworth, A.K.,Consistency in Networks of Relations, *Artificial Intelligence* **8** (1977), pp.99-118.

[12] Mackworth, A.K. and Freuder, E.C., The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems, *Artificial Intelligence* **25** (1985) pp.65-74.

[13] Mohr, R. and Henderson, T.C., Arc and Path Consistency Revisited, *Artificial Intelligence* **28** (1986) pp.225-233

[14] Meseguer, P.,Constraint Satisfaction Problems: An Overview, *AICOM Vol.2 No.1* 1989, pp.3-17.

[15] Nudel, B., Consistent Labeling Problems and their Algorithms: Expected Complexities and Theory-Based Heuristics, *Artificial Intelligence* **21** (1983) pp.135-178.

[16] Yaglom, A.M., and Yaglom, I.M., Challenging Mathematical Problems with Elementary Solutions. *Holden-Day, San Fransisco*, 1964.