Exploitation of Interchangeabilities in Search

Amy Beckwith
Department of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE 68508
abeckwit@cse.unl.edu

December 12, 2000

Abstract

Interchangeabilities among values of a variable in a Constraint Satisfaction Problem can be detected and are useful to make search more effective when solving CSPs. This paper discusses the exploitation of interchangeabilities and their effects on search. It also shows classes of problems where search does not benefit from interchangeability.

1 Introduction

The primary method of solving a Constraint Satisfaction Problem (CSP) is by search. Therefore, we are constantly looking for better and more effective ways to perform search. For any problem the search space may be up to d^n in size, where d is the size of the largest domain, and n is the number of variables. To make search faster, we wish to reduce this search space as much as possible.

One way to reduce the search space is to find characteristics of the problem. Here we focus on finding interchangeable values within a variable—which values will never participate in a solution, or which values are redundant. Such values can then be removed, leaving a trimmed search space for a faster search. In this paper we compare the cost of finding such values with the gain experienced by removing them from the search space.

By finding interchangeable values, we can not only reduce the size of a search space, but we can find readily groups of values which will all either participate together in a solution, or will not participate in any solution. We can also use these groups of interchangeable values to find similarities between sets of solutions and to group similar solutions together.

Finding such interchangeabilities in a CSP is not trivial. In this paper, I shall first introduce several types of interchangeabilities and how to compute them. Then I shall discuss how to make such interchangeabilities useful while conducting search. I shall introduce experiments and empirical results of finding interchangeabilities and their utility in search. Finally, I shall conclude with directions for future research on this topic.

2 Background

Within a Constraint Satisfaction Problem, there are many types of Interchangeabilities which may be found. The interchangeabilities discussed here were formally defined by Freuder [3] and extended by Haselböck [4] and Choueiry [2].

The strongest and most global kind of interchangeability is called Full Interchangeability (FI). A value b for a CSP variable V is fully interchangeable with a value c for V iff every solution to the CSP which contains b remains a solution when c is substituted for b and vise versa. The only difference between these two solutions is the values b and c. Everything else remains the same, and both are solutions. Therefore, these two values could be replaced by one value. By replacing these two values by a representative, we keep the search algorithm from generating exactly the same tree for both values and we do not lose any information. However FI is hard to find, for it requires generating all solutions and looking for their similarities. This is not practical, so instead we find interchangeabilities on a local level.

With Neighborhood Interchangeability (NI), we limit interchangeability within a variable to involve only other variables linked to it with a constraint—its neighborhood. A value b for a CSP variable V is Neighborhood Interchangeable with a value c iff for every constraint C of V, b and c are consistent with exactly the same values, as stated in Equation (1).

$$\{x|(b, x) \text{ satisfies } C\} = \{x|(c, x) \text{ satisfies } C\} \tag{1}$$

By considering a portion of the problem rather than the whole CSP, NI finds interchangeabilities that are also FI, but at a much lower cost. Instead of computing all solutions, one must merely iterate over the constraints of variable V. Freuder [3] proves that this can be done in $O(n^2d^2)$, and gives an algorithm. NI looks at the effect of one variable on its immediate environment, rather than on the problem as a whole. For example, in Figure 1 (a), computing the NI partition for $V2$ requires considering the constraints in bold and ignores the effects of the constraints shown with a dotted line.

We can find further interchangeabilities by weakening the requirement to allow some change. Partial Interchangeability (PI) allows some values to change among themselves while the rest of the problem remains constant. Therefore, we are finding a portion of the problem in which change is acceptable. Two values are Partially Interchangeable with respect to a subset S of all the variables iff a solution involving one of the values implies that there is a solution with the other value with changes only inside of S. S delimits the boundary of change. In Figure 1 (b), this boundary of change is a curve around the enclosed variables. Like FI, PI requires calculation of all solutions, but values that are PI are not necessarily FI. Again we find that PI on a local level is easier to calculate.

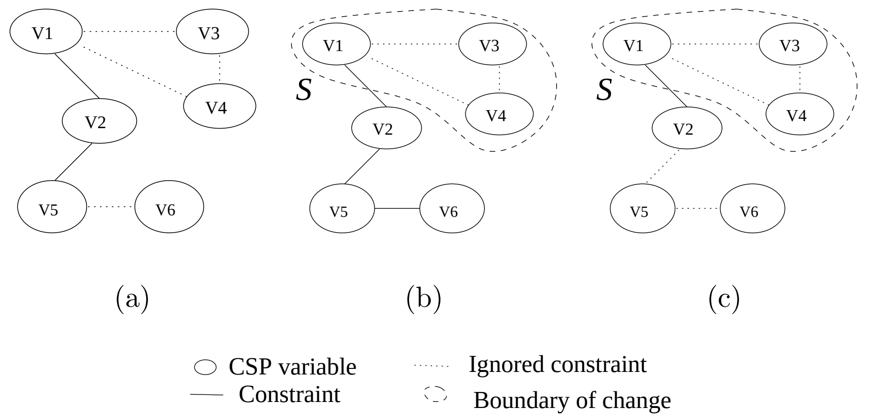

Figure 1: *NI, PI and NPI.* (a) NI for variable $V2$ ignores constraints not incident to $V2$. (b) PI for subset S ignores constraints that are within S. (c) NPI for subset S and constraints not incident to variables in S.

Neighborhood Partial Interchangeability (NPI), like NI, takes a subset S of all variables and finds the interchangeabilities ignoring constraints among those variables. However, unlike NI, NPI only considers constraints that are incident to one variable inside S and one variable outside of S. [2] In order to find NPI for a variable V which is inside S, we make sets of the values in V according to values for variables outside S with which they are consistent. If two values are consistent with exactly the same values for every constraint in V that crosses S, they are NPI. In Figure 1 (c), we see that this considers only constraints in the Neighborhood of S, and ignores all others.

We can find even more interchangeability if we further weaken this requirement. Instead of finding interchangeabilities among groups of variables, Haselböck [4] suggests that we find interchangeabilities across a single constraint. This Neighborhood Interchangeability across a Constraint (NI_C) is the weakest and most pervasive kind of interchangeability yet. A value a is NI_C with a value b with respect to a constraint C iff for that constraint a and b are consistent with exactly the same values in the other variable of the constraint, as stated in Equation (2).

$$\{x|(b,x) \text{ satisfies } C\} = \{x|(c,x) \text{ satisfies } C\}. \tag{2}$$

By finding NI_C, we can partition the domain of a variable into sets that are consistent with the same values in one other variable. Because it goes across only one constraint, we can calculate NI_C in $(2d)^2\dot{e}$, where d is the domain size and e is the number of constraints in the CSP. One must merely find with what values in the other variable across the constraint each value is consistent, and combine them into sets. Note that this is directional. If I am computing the NI_C of V_i with respect to a constraint between V_i and V_j, it is not the same as computing the NI_C of V_j with respect to the same constraint. These must be computed separately.

As one can see in Figure 2, Finding an NI_C for a variable V across a constraint C is equivalent to taking NPI with the boundary of change S being drawn around the neighborhood of V excluding C.

NI_C is not only easy to compute, but we can use it to find NPI or NI. A variable $V1$ in a graph has a certain number of constraints connecting it to other variables. NI_C computes the interchangeability across only one of these constraints. If we were to compute the NI_C across another of those constraints, then we could combine the two constraints and partition the domain of $V1$ by intersecting the partitions from the two constraints. We then would have divided the constraints into two groups, ignoring all but two constraints, and computing the NPI across the two. We can further extend the NPI that we compute until we have taken the intersection of all the constraints in $V1$, then we have computed NI. NI can be used to approximate FI. Therefore, by intersecting NI_C sets, we can find other interchangeabilities.

3 Search with Interchangeabilities

When the interchangeabilities of a problem can be discovered, they can be used to reduce a search space. Haselböck [4] describes an algorithm for search that exploits known NI_C. We adapted his algorithm for our purposes. In Figure 3, we show how our new Revise works.

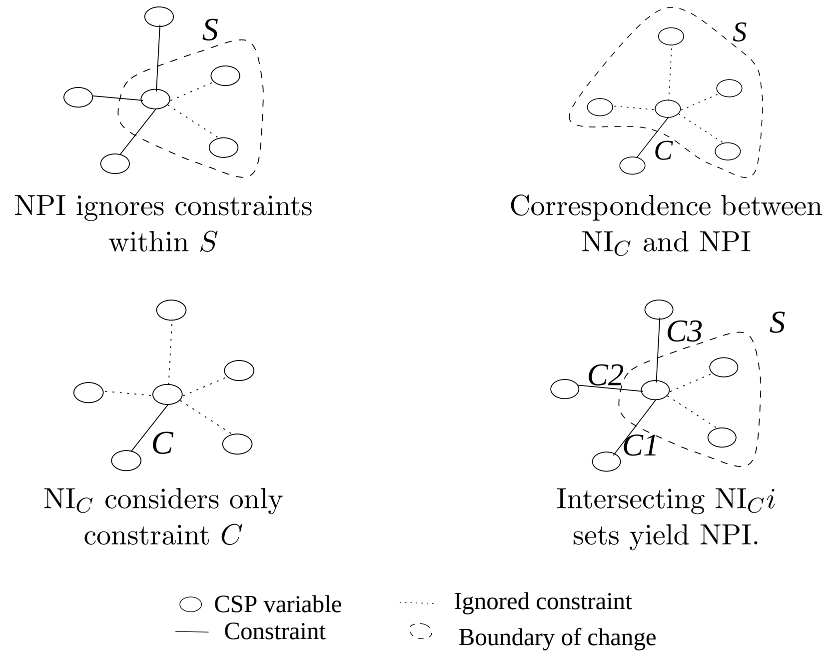

Figure 2: *NI_C and its relation to NPI*

Algorithm 1 *Adapted-Revise($V1$, C):*

Begin
Partition the domain of $V1$ according to C.
Let $V2$ be the variable incident to C
Let v2 be one representative of the assignment bundle of $V2$
For every set in the partition of the domain of $V1$
  Let v1 be one representative of the values in the considered set of $V1$
  If v1 and v2 are consistent
    Then Move to the next partition of the domain $V1$
    Else remove the entire set containing $v1$ from the domain of $V1$
End

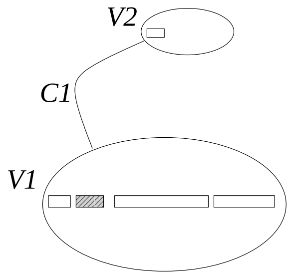

Figure 3: *Adapted from Haselböck's revise procedure.* Revise the domain of one variable according to the other.

With this revise, we perform search. Figure 4 shows the Back-Tracking (BT) search, where we revise the current variable according to past assignments. In Forward-Checking (FC), we instead Revise the domain of each future variable according to the assignment to the current variable.

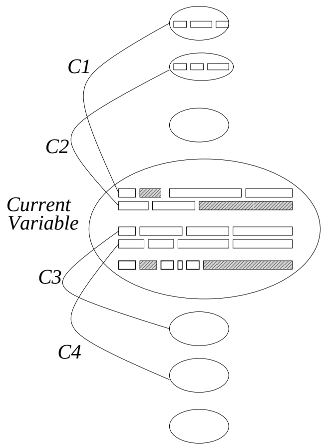

Figure 4: *Adapted from Haselböck's search algorithm*

Algorithm 2 *Node-expansion in BT-Search with Interchangeabilities (V):*

Begin
Separate constraints incident to V into past and future constraints.
For every constraint C_p with past variable
    Update the domain of V using Adapted-Revise(V, C_p)
Then partition the updated domain of V by making the intersection of the partitions of V according to <u>all</u> future constraints
Assign one of these partition groups to V.
End

Algorithm 3 *Node-expansion in FC-Search with Interchangeabilities (V):*

Begin
Separate constraints incident to V into past and future constraints.
Partition the domain of V by making the intersection of the partitions of V according to <u>all</u> future constraints
Assign one of these partition groups to V.
Then, For every constraint C_f with future variable $V2$
        Update the domain of $V2$ using Adapted-Revise(V, C_p)
End

If one takes a careful look at the search routine, they will see that each NI_C set is used exactly once during the instantiation of a variable. To see this, consider again a variable V_i. When V_i is the current variable in the search, we partition the domain of V_i according to one of the constraints to a past variable, and prune the domain accordingly. We repeat this for each of the past variables. Therefore, if C_i is a constraint to a past variable, the NI_C for that constraint will be tested once here. Then we re-partition the domain of V_i according to the future variables. This requires testing the NI_C of V_i with respect to each of the future variables exactly once also. It is important to remember that NI_C is directional and therefore the partition of V_i with respect to a constraint between V_i and V_j is not the same NI_C as the partition of V_j with respect to the same constraint. So, if a constraint is checked from V_i to a future variable V_j when we are partitioning the domain for V_i for assignment, that same NI_C will not be computed when we are at V_j and checking against the past to V_i. Because of this directionality, the NI_C across each constraint is checked once and only once each time we expand a variable.

This does not mean that the NI_C is only checked once overall, for if we backtrack, some variables will be expanded more than one time, and consequently their NI_C sets will also be used multiple times.

4 Contributions

In this project, I implemented all of the data structures necessary for CSPs in Lisp. Using these data structures, I implemented several CSPs. I also implemented a full back-tracking search algorithm and a forward-checking search algorithm, both according to Prosser [6]. Next, I adapted Choueiry and Noubir's [2] algorithm and Choueiry's code for computing NPI in order to find to find NI_C. Finally, I implemented search strategies for back-tracking and forward-checking that exploit these interchangeabilities and compare them to classical back-tracking and forward-checking in several kinds of test-problems. Through these tests, I find classes of problems where search does not benefit from the NI_C. I also explore ideas for future use of NI_C in my research.

Currently in my work, and for this project, NI_C is computed once, at the beginning of the search. However, as I show above, these NI_C sets are not frequently used. Therefore it is conceivable that we may want to re-compute the NI_C every time we want to use an NI_C set rather than storing them. To do such a thing would allow us to find Dynamic Interchangeabilities.

Dynamic Interchangeability (DI) is an interchangeability that can be recalculated as search takes place. As we make choices for some variables and prune the domains of others, more interchangeability may surface. A value that keeps two other values from being interchangeable may be removed from a domain, leaving more interchangeability in some other variable. This DI can perhaps be exploited by computing NI_C every time we instantiate a variable rather than at the beginning of the search.

Computing NI_C before starting search is equivalent to computing NI_C using the original domains of the variables. As we search, the current domains of the variables change. Rather than compute the NI_C before beginning search, we can postpone computation until the NI_C sets are actually needed, exploiting DI. This will begin to break down if there is a large amount of back-tracking that takes place. In this case NI_C will have to be re-computed every time a node is expanded, since the current domains of some variables will be different. The actual utility of DI will need to be verified experimentally.

The experiments that I conducted do not explore nor exploit DI. Rather, they compute NI_C statically, and use the NI_C sets generated at the beginning of the problem whenever necessary.

5 Experiments

The experiments I chose are as shown in Table 1:

Problem	# vars	Domain size	# const.	Const. density	Const. tightness
3-queens	3	3	3	1	0.703704
4-queens	4	4	6	1	0.541667
5-queens	5	5	10	1	0.44
6-queens	6	6	15	1	0.37037
7-queens	7	7	21	1	0.319728
8-queens	8	8	28	1	0.28125
zebra	25	5	64	0.137681	0.956129
ra 1	25	6	170	0.528986	1
ra 2	35	7	261	0.404635	1
ra 3	36	7	306	0.455462	1
ra 4	25	5	139	0.416667	1
random*	10	5	13 to 41	0.1 to 0.9	0.11 to 0.99

Table 1: *Problems chosen for comparison.* *For random problems, five instances of the problem were generated with each of the following: constraint density = 0.1, 0.5 and 0.9, which is 13, 27 and 41 constraints, respectively. constraint tightness = 0.11, 0.33, 0.55, 0.77, 0.99. this results in 75 problems, which were then averaged for the results reported.

This includes three classes of problems - puzzles, real-world resource allocation, and random problems. All of these were solved using four search algorithms: back-tracking search (bt-search), forward-checking search (fc-search), back-tracking that exploited NI_C (h-bt-search), and forward-checking that exploited NI_C (h-fc-search). Each of these four searches found not only one solution, but all solutions, that we might see the benefit of grouping similar solutions together allowed by finding interchangeabilities.

Random problems were generated using Bacchus' random generator [1]. His code generates random CSPs given the number of variables (N), the maximum domain size for a variable (A), the number of constraints (C), and the number of disallowed tuples per constraint (T).

To compare the searches we counted the number of nodes expanded by each search routine and the number of constraints checked during the search. All problems are reported with the number of constraint checks needed to compute NI_C added to the constraints checked during search. Ideally, we will also compute the time that each algorithm takes while solving the problems.

6 Results

Puzzles As shown in Table 2 (and for the other queens problems), NI_C costs more to compute than it saves during search in the N-queens problems. Additionally, no solution bundles can be found from computing the NI_C sets, so by computing NI_C, we are not gaining any information about the problem.

search	Nodes expanded	Const. tested	h. const. tested	# solu. bundles
bt-search	6241	72240		92
h-bt-search	4712	62574	7168	92
fc-search	1780	15492		92
h-fc-search	1766	14844	7168	92

Table 2: *8-queens results: 92 solutions, using variable ordering: q4 q8 q1 q5 q2 q6 q3 q7* Note: tests were run on 3-queens to 8-queens. The tables are similar but not shown in order to conserve space.

Similarly, in Table 3 we see that finding the NI_C sets obviously hurts the performance of the search on the zebra problem. The number of constraints tested due to calculating the NI_C sets are not made up in search. In this problem case, there are no solution groups to have located.

search	Nodes expanded	Const. tested	h. const. tested	# solu. bundles
bt-search	888	6174		1
h-bt-search	592	3567	3112	1
fc-search	144	1567		1
h-fc-search	138	1512	3112	1

Table 3: *Zebra results: 1 solution, using variable ordering: MILK BLUE NORWEGIAN GREEN IVORY COFFEE TEA ENGLISHMAN RED UKRAINIAN JAPANESE SPANIARD WATER LUCKY-STRIKE KOOLS ORANGE YELLOW DOG HORSE PARLIAMENT ZEBRA FOX OLD-GOLD SNAILS CHESTERFIELD*

Resource-Allocation Problems Even in resource allocation, NI_C does not benefit us in search. The cost of computing the NI_C sets for this resource allocation problem is considerably larger than the gain due to exploitation of interchangeabilities in the search, as shown in Table 4.

search	Nodes expanded	Const. tested	h. const. tested	# solu. bundles
bt-search	1244	9963		0
h-bt-search	860	7096	5076	0
fc-search	395	5286		0
h-fc-search	380	4442	5076	0

Table 4: *Resource Allocation results: 0 solutions using variable ordering: (OP22 OP8 OP4 OP1 OP13 OP10 OP12 OP19 OP15 OP16 OP6 OP9 OP23 OP5 OP2 OP14 OP17 OP20 OP7 OP25 OP11 OP24 OP18 OP21 OP3)* Note: Four data-sets were tested, with similar results. The tables are not shown here in order to conserve space.

Random Problems However in Random problems, we begin to see much more benefit from Interchangeability. This is because, in random problems, we can easily control the constraint density and tightness in order to better see under which conditions search will be able to find and exploit interchangeabilities. In Figure 6, we see that in all cases, Back-Tracking while taking advantage of interchangeabilities expands fewer nodes than does backtracking without interchangeabilities. We also see that the same comparison holds true for the forward-checking searches. A search the exploits interchangeabilities expands fewer nodes than does the search without this knowledge.

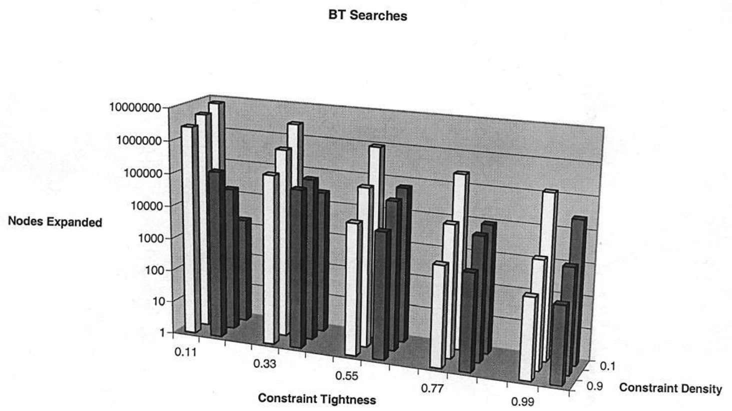

Figure 5: *Nodes expanded by BT-search (light) compared to Nodes expanded by H-BT-search (dark).*

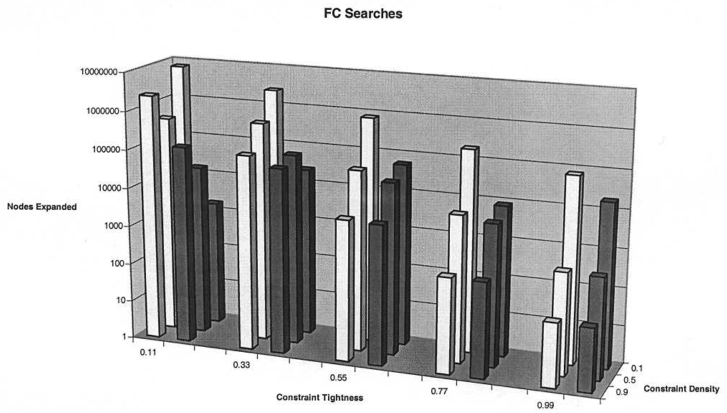

Figure 6: *Nodes expanded by FC-search (light) compared to Nodes expanded by H-FC-search (dark).*

Similarly, we see in Figure 8 that searches that take advantage of interchangeability usually test fewer constraints than do their counterpart that does not exploit interchangeability. Notable exceptions are, are you can see, when constraints are very tight. In this case, expecially in forward-checking search.

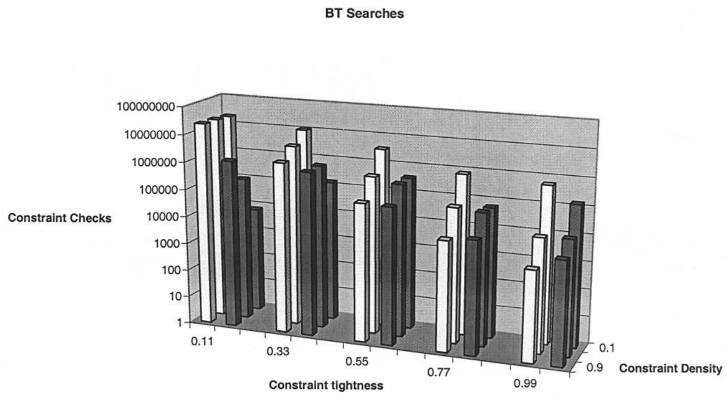

Figure 7: *Constraints tested by BT-search (light) compared to constraints tested by H-BT-search (dark).*

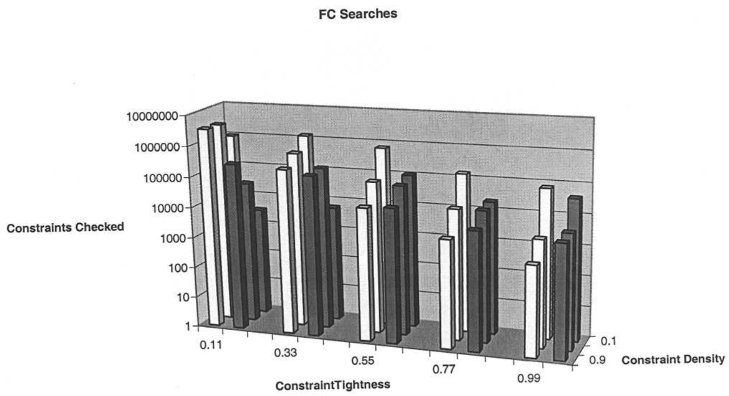

Figure 8: *Constraints tested by FC-search (light) compared to constraints tested by H-FC-search (dark).*

This can also be viewed as related to the number of solutions, rather than to just the constraint tightness and density. In Figure 10, we can see that in the places where we saw that search took advantage of interchangeability the best (low constraint density, and low constraint tightness), we have many more solutions to the problem. We can also see that with this large number of solutions, the searches which exploit interchangeability step up to the job and find bundles of solutions. Figure 10 shows the sizes of the solution bundles found by the searches, and the bundle size is accordingly large where the most solutions were found.

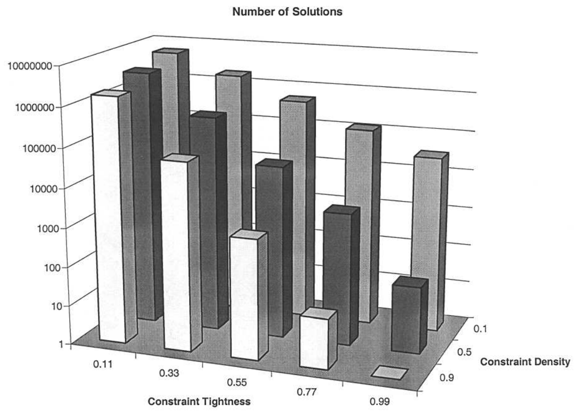

Figure 9: *Number of solutions found in random problems.*

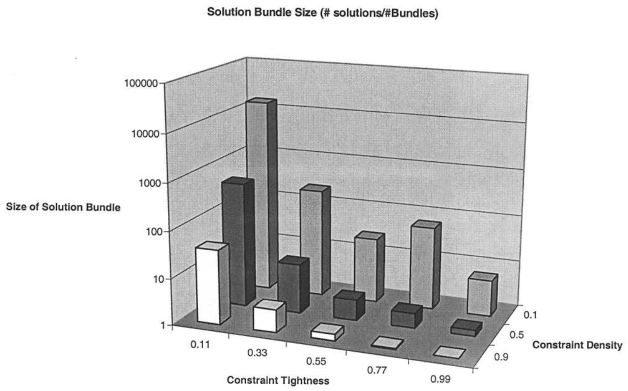

Figure 10: *Size of solution bundles found by searching with interchangeability in random problems. (# Solutions/ # Solution Bundles)*

Obviously search is at a great advantage from having the NI_C sets in almost all of these random problems. Even when the cost of computing the NI_C sets is added, there is a significant gain in terms of nodes expanded, constraints tested, and in a much-reduced solution size. This shows that indeed the NI_C sets can be beneficial to search.

7 Conclusions and Future Work

We found that in the n-queens problem, making use of NI_C sets does not provide benefit. While NI_C does exist, it is not frequent enough to overcome the cost of computation of the NI_C sets. In addition to this, very little NPI can be found in the n-queens problems, so all solutions were in sets of size one. This is to be expected, considering that n-queens problems yield a complete constraint graph—they contain the maximum number of constraints. Therefore the variables interact very significantly, and limit the interchangeability present.

Likewise in the Zebra problem, there was even less benefit from the NI_C sets. This is because, in the zebra problem, there is no NI_C. Rather, in the zebra problem, are five graph-coloring cliques of size five. These five cliques are linked together by functional constraints such that for each value in one variable, the acceptable values in the other variable differ by at least one. For example, a constraint may say that the person who smokes Chesterfield must live next to the person who owns the dog. In this case, the micro-structure of this constraint would look like as shown in Figure 11.

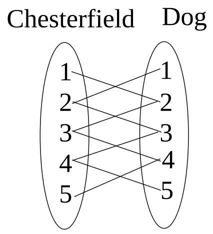

Figure 11: *Micro-structure of a zebra constraint.*

Upon inspection, one will see that every constraint in the zebra problem is of this nature. Therefore, there can be no interchangeability across any of the constraints. Because of the same nature, there is no NPI in the problem. Computing NI_C sets brings no benefit, because we can never use them.

This shows an important characteristic of puzzles. They are formulated to be tricky and difficult. The design of them shows up on the lack of interchangeabilities that can be found. For CSPs with low or no interchangeability, it is not profitable to compute even NI_C to aid in search. Such problems include common puzzles.

Resource allocation in this case does not fare much better than the puzzles. However, I suspect that this will not be the general case. In resource allocation problems where the resources are not quite so limited, or where there actually exists a solution, we may find that more interchangeabilities can be found and exploited.

Random problems give us an explanation for the behavior of the other problems. From the random-problems charts, we see that CSPs with tight constraints, such as n-queens, are not likely to have much interchangeability in any sense. Problems with a low constraint tightness seem to benefit much more from interchangeability. As constraints become tighter, there is less interchangeability overall, so the cost of computing NI_C is not worthwhile. This follows intuition, for constraint tightness represents allowed pairs of values that two variable scan take. As fewer pairs are allowed, interchangeabilities will decrease. This same idea can be observed, though less so, with the constraint density. More constraints depict higher levels of interaction between variables, and thus fewer opportunities for interchangeability. When there is little or no interchangeability, search cannot benefit from it, and the cost of computing the interchangeabilities is wasted. With loose constraints and few constraints, there are many solutions to the CSP. In these, there is likely to be a high amount of interchangeability, and the solutions can be grouped in to rather large bundles.

Constraint tightness and constraint density can likely be used as a test. If the constraint density is high, and the constraints are tight, the cost of finding interchangeabilities in a problem, even interchangeability as cheap as NI_C, will not prove useful in search.

Nevertheless, search that exploits NI_C will expand as few or fewer nodes than searches that do not exploit interchangeability, and the cost of calculating these interchangeabilities is fixed and linear. For CSPs with loose constraints, and without a large number of constraints, interchangeabilities are valuable. Even a simple algorithm such as the one used, with static variable and value ordering, was able to detect a large number of interchangeabilities, save many constraint checks and bundle solutions together. These are the goals of finding interchangeability, and with NI_C, they are easy to meet.

In the future, I intend to implement DI using NI_C and compare it to both static NI_C in search and to search with Cross-Product Decomposition [5] in search. I expect that we will see DI out-perform both static NI_C and search according to Cross-Product Decomposition. However, many tests will need to be done.

Acknowledgments

Thanks to Berthe Choueiry, who directed me and allowed me to use her code for computation of interchangeabilities.

References

[1] Fahiem Bacchus and P. van Run. Dynamic Variable Ordering in CSPs. In *Principles and Practice of Constraint Programming, CP'95. Lecture Notes in Artificial Intelligence #976, Springer Verlag.*, pages 258–275, 1995.

[2] Berthe Y. Choueiry and Guevara Noubir. On the Computation of Local Interchangeability in Discrete Constraint Satisfaction Problems. Technical Report KSL-98-24, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, Stanford, CA, 1998. Premilinary versions appeared in the Proceedings of AAAI'98.

[3] Eugene C. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proc. of AAAI-91*, pages 227–233, Anaheim, CA, 1991.

[4] Alois Haselböck. Exploiting Interchangeabilities in Constraint Satisfaction Problems. In *Proc. of the 13 th IJCAI*, pages 282–287, Chambéry, France, 1993.

[5] Paul D. Hubbe and Eugene C. Freuder. An Efficient Cross Product Representation of the Constraint Satisfaction Problem Search Space. In *Proc. of AAAI-92*, pages 421–427, San Jose, CA, 1989.

[6] Patrick Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9 (3):268–299, 1993.