Solving Non-Binary CSPs Using the Hidden Variable Encoding

Nikos Mamoulis[1] and Kostas Stergiou[2]

[1] CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
nikos@cwi.nl,
[2] University of Glasgow, Department of Computer Science, Scotland
kostas@dcs.gla.ac.uk

Abstract. Non-binary constraint satisfaction problems (CSPs) can be solved in two different ways. We can either translate the problem into an equivalent binary one and solve it using well-established binary CSP techniques or use extended versions of binary techniques directly on the non-binary problem. Recently, it has been shown that the hidden variable encoding is a promising method of translating non-binary CSPs into binary ones. In this paper we make a theoretical and empirical investigation of arc consistency and search algorithms for the hidden variable encoding. We analyze the potential benefits of applying arc consistency on the hidden encoding compared to generalized arc consistency on the non-binary representation. We also show that search algorithms for non-binary constraints can be emulated by corresponding binary algorithms that operate on the hidden variable encoding and only instantiate original variables. Empirical results on various implementations of such algorithms reveal that the hidden variable is competitive and in many cases better than the non-binary representation for certain classes of non-binary constraints.

1 Introduction

The majority of the research on constraint satisfaction problems (CSPs) has focused on algorithms and heuristics that are applied on binary problems. The main reason for this is that any problem that contains constraints of an arbitrary arity can be transformed to an equivalent binary problem [11]. In the past, research on non-binary CSPs has mainly dealt with filtering algorithms. Recently, it is being recognized that more research on other non-binary issues is also required. As a result, search algorithms for binary CSPs have been extended for non-binary ones ([3]) and the efficiency of binary encodings has been investigated ([1, 12, 7]).

The most popular binary translations are the dual graph encoding and the hidden variable encoding. It is not clear which of the two is the best. However, the hidden variable encoding has some nice theoretical properties which make it a promising technique in many cases [12, 13]. First, arc consistency (AC) on this binary representation achieves the same consistency level as generalized arc consistency (GAC) on the non-binary problem. This means that MAC (i.e., *maintaining arc consistency*) applied on the hidden variable encoding of a non-binary CSP visits the same search tree nodes as MGAC (i.e., *maintaining generalized arc consistency*) on the non-binary representation. Second, enforcing AC on an arbitrary encoded non-binary constraint takes the same number of consistency checks in the worst-case as GAC on its non-binary representation. These theoretical results, indicate that the hidden variable encoding is a promising way of solving non-binary CSPs with MAC. In practice, we can only use the hidden variable encoding on CSPs that have tight constraints. For CSPs with a large number of loose constraints it is reasonable to assume that the hidden variable encoding will be inefficient due to the large space requirements. It has also been shown experimentally that solving the binary encoding of a non-binary CSP can be less efficient than applying a non-binary version of some search algorithm, and vice versa, depending on the tightness of the constraints [1, 12].

In this paper we take a closer look on arc consistency and search algorithms for the hidden variable encoding. The difference between an arc consistency algorithm on the encoding and a generalized arc consistency algorithm is the fact that the former has to update the domains of the hidden variables as well as the original ones. We show that this can lead to an arc consistency algorithm that runs on the encoding and, for any arc consistent graph, performs exactly the same number of consistency checks as the corresponding generalized arc consistency algorithm. For arc inconsistent graphs we show that the AC on the encoding can detect the inconsistency earlier and thus perform fewer checks than GAC. In a special case, the algorithms are equivalent not only in consistency checks but also in all the primitive operations they perform (e.g. domain lookups and deletions). In general, there is a trade-off between the binary and non-binary algorithms in the amount of primitive operations they perform. We also show that, like MGAC, the generalizations of forward checking to non-binary CSPs can be simulated by a corresponding binary forward checking algorithm on the hidden variable encoding that only instantiates original variables, resulting in the same node visits. We make an empirical comparison of different implementations of binary and generalized algorithms which reveals that the hidden variable encoding can be competitive and often better than the non-binary representation in certain classes of tight non-binary CSPs.

2 Background

A *constraint satisfaction problem* (CSP) $\mathcal{P}$ is defined by a triple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$. $\mathcal{X}$ is a set of n variables. Each variable $x_i \in \mathcal{X}$ takes values from a domain $D_i \in \mathcal{D}$. $\mathcal{C}$ is a set of e constraints. Each k-ary constraint is defined over an ordered set of variables $\{x_1, \ldots, x_k\}$ by a subset of the Cartesian product $D_1 \times \ldots \times D_k$ that specifies the set of allowed value combinations (tuples). A constraint can be defined either extensionally by the set of allowed tuples or intensionally by a predicate or arithmetic function. In the following we will assume that all non-binary constraints are defined extensionally by nature, or can be represented extensionally without excessive space requirements. We also assume that there is at most one constraint per variable combination.[1]

A value a in the domain D of variable x is *consistent* with a constraint c if x is not included in the variables of the constraint, or if it is included and there exists a valid tuple τ in c where $x = a$. In the latter case we say that τ is a *support* for a in c. Checking whether a tuple is a support for a variable value pair (x, a) is called a *consistency check*. A variable x is *consistent* with a constraint c if $D \neq \emptyset$ and all its values are consistent with c. A constraint c is *arc consistent* (AC) if $\forall x_i \in \mathcal{X}$, x_i is consistent with c. A binary CSP is arc consistent if all its constraints are arc consistent. A CSP is *singleton arc consistent* (SAC) iff it has non-empty domains and for any instantiation of a variable, the problem can be made arc consistent. We call the generalizations of AC and SAC to non-binary CSPs GAC and SGAC respectively. Finally, a solution to a CSP is an assignment of values to variables which are consistent with all constraints.

Following [8], we call a local consistency property A *stronger* than B iff for any problem A deletes at least the same values as B, and *strictly stronger* iff it is stronger and for at least one problem A deletes more values than B. We call A *equivalent* to B iff they delete the same values for all problems. Similarly, we call a search algorithm A stronger than an algorithm B iff for every problem A visits at most the same search tree nodes as B, and strictly stronger iff it is stronger and for at least one problem A visits less nodes than B. A is equivalent to B iff they visit the same nodes for all problems.

2.1 Hidden Variable Encoding

The *hidden variable encoding* [11] is a well-known method for transforming a non-binary CSP to a binary one. It encodes the non-binary constraints to variables (called "hidden" variables) that have as domain the valid tuples of the constraint. For each tuple in the domain of the hidden variable v_c, the encoding introduces compatibility constraints between v_c and each original variable x_i in the constraint c. Each constraint specifies that the tuple assigned to v_c is consistent with the value assigned to x_i. Consider the following example with six variables with 0,1 domains, and four constraints: $x_1+x_2+x_6 = 1$, $x_1-x_3+x_4 = 1$, $x_4 + x_5 - x_6 \geq 1$, and $x_2 + x_5 - x_6 = 0$. In the hidden variable encoding (Figure 1) there are, in addition to the original six variables, four hidden variables. The domains of these hidden variables are the tuples that satisfy the respective constraint. For example, the hidden variable associated with the third constraint v_3 has the domain $\{(0,1,0),(1,0,0),(1,1,0),(1,1,1)\}$, as these are the tuples of values for (x_4, x_5, x_6) which satisfy $x_4+x_5-x_6 \geq 1$. There are now compatibility constraints between v_3 and x_2, between v_3 and x_5 and between v_3 and x_6, as these are the variables mentioned in the third constraint.

[1] Multiple constraints on the same set of variables can be reduced to a single constraint in the extensional representation.

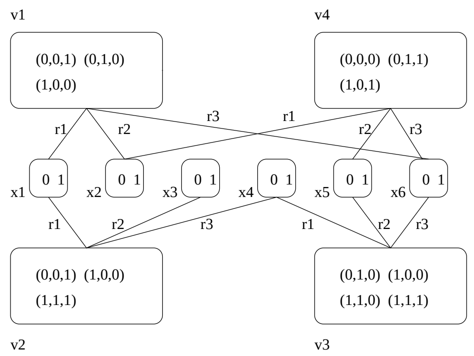

Fig. 1. Hidden variable encoding of a non-binary CSP. The binary constraint ri applies to a tuple and a value and is true iff the ith element of the tuple equals the value.

3 Arc Consistency

In this section we study the relationship between AC on the hidden variable encoding and GAC in more detail by examining the benefits of revising the domains of hidden variables. We will show that these revisions can help an AC algorithm on the encoding to identify inconsistencies earlier than the corresponding GAC algorithm.

3.1 GAC Algorithms

GAC-4 [10] is designed for constraints represented in extension by their allowed tuples. Each time a value a is deleted from a variable x, the tuples that include this variable-value pair are also deleted from the lists of allowed tuples. The deletion of these tuples may trigger the deletion of further values that lose their support, and so on. We can view this algorithm as a binary algorithm that runs on the hidden variable encoding. The only modification we need to make is to consider a constraint c as a hidden variable h_c and the set of allowed tuples of c as the domain of h_c. The propagation of deletions can then be done in exactly the same way resulting in the same primitive operations as in the non-binary case.[2] By *primitive operation* we mean a domain lookup (i.e, check if a value is in the domain of a variable), a deletion of a value (or a tuple), a consistency check, and any other check in a list or other data structure.

GAC-3 is an extension of the well-known AC-3 algorithm to non-binary CSPs. When a value is deleted from a variable, GAC-3 adds to a stack all constraints that involve that variable. Then, constraints are removed from the stack and are "revised". Revising a constraint means searching for a new supporting tuple for the values of all variables in the constraint. Checking whether an variable-value assignment is consistent with respect to a constraint $c = (x_1, \ldots, x_k)$ involves

[2] This equivalence has been pointed out by Christian Bessière at CP'99.

finding all tuples $< a_1, \ldots, a_k >$ in c that contain this assignment and checking if values $a_1, \ldots, a_k$ are still in the domains of variables $x_1, \ldots, x_k$. The reason for this is that GAC-3 like algorithms, in their standard implementation, do not make updates in the lists of allowed tuples like GAC-4 does when a value is deleted. So, they cannot check directly if tuple $< a_1, \ldots a_k >$ is still valid. This results in extra operations compared to GAC-4, but on the other hand GAC-3 like algorithms avoid updating the usually large sets of allowed tuples (i.e., hidden variable domains) and require less space. Like GAC-4, a GAC-3 algorithm that updates the lists of allowed tuples can be viewed as a binary algorithm that operates on the hidden variable encoding. GAC-schema [5] is another GAC algorithm that does not update the allowed tuples but instead looks for supports in a similar, but more sophisticated, way as GAC-3.

Recently, the binary AC-3 algorithm has been modified to yield an algorithm with optimal worst-case time complexity [6, 14]. What makes the new AC-3 algorithms optimal is the use of a pointer $currentSupport_{x,a,c_{xy}}$ for each value a of a variable x involved in a constraint c between x and y. This pointer records the current value in the domain of y that was found to be a support of a. After a value deletion, if we look for a new support for a in y, we first check if the value where $currentSupport_{x,a,c_{xy}}$ points is still in the domain of y. If not, we search for a new support starting from the value immediately after the current support. Assuming that the domains are ordered, [6, 14] prove that the new algorithm is optimal. This algorithm can be extended to non-binary constraints in a straightforward way. Again, we can use a pointer $currentSupport_{x,a,c}$ that points to the last tuple (assuming an ordering of the tuples) in constraint c that supported value a of variable x, where x is a variable involved in c. A sketch of the main functions of the algorithm, omitting the initialization phase, is shown in Figure 2. We now briefly discuss the complexity of this algorithm.

Like GAC-3, when a variable-value pair (x, a) is deleted, each constraint involving x is pushed on the stack. Then, constraints are popped from the stack and revised. Each $k-$ary constraint can be revised at most kd times, one for every deletion of a value from the domain of one of the k variables. Since we use the pointers $currentSupport_{x,a,c}$, for each variable-value pair (x, a) we can check at most d^{k-1} subtuples to find a support.[3] This results in $O(kdd^{k-1})$ checks for one constraint in the worst-case. For e constraints the worst-case complexity, measured in consistency checks, becomes $O(ekd^k)$. To check if a tuple is valid, in lines 3 and 4, we have to check if the values in the tuple are present in the domains of the corresponding variables. If one of these values has been deleted then the tuple is not valid.

3.2 AC on the hidden variable encoding

As discussed, the worst-case cost of AC on the hidden variable encoding, measured in consistency checks, is the same as GAC on the non-binary representa-

[3] In fact, $\min\{d^{k-1}, |T|\}$ subtuples, where $|T|$ is the number of allowed tuples in the constraint. See [6, 14] for details.

function *Propagation*
 While Q is not empty
  pick c from Q
  for each uninstantiated $x_i \in c$
   if $Revise(x_i, c) = TRUE$ *then*
    if domain of x_i is empty **then return** INCONSISTENCY
1   put in Q all constraints that involve x_i
 Return CONSISTENCY
function $Revise(x_i, c)$
 DELETION $\leftarrow$ FALSE
  for each value a in the domain of x_i
2   **if** $currentSupport_{x_i,a,c}$ is not *valid* **then**
3   **if** $\exists\ \tau (\in\ c) >\ currentSupport_{x_i,a,c}$, τ includes (x_i, a) and τ is *valid*
     then $currentSupport_{x_i,a,c} \leftarrow \tau$
4   **else** remove a from the domain of x_i
    DELETION $\leftarrow$ TRUE
 Return DELETION

Fig. 2. The algorithm of [6, 14] for non-binary CSPs.

tion. When GAC-4 and its equivalent in the encoding are used, we can also get exactly the same number of primitive operations. We now analyze the difference between the extended GAC-3 algorithm and its equivalent on the encoding. To get the hidden variable equivalent of the GAC-3 algorithm shown in Figure 2 we need to make 3 changes. First, any references to constraints are substituted by references to hidden variables. For example, line 1 in Figure 2 will read: "put in Q all hidden variables that involve x_i". Second, after a value is removed from the domain of an original variable (line 4), all tuples that include that value are removed from the domains of the corresponding hidden variables. Third, checking if a tuple is *valid* is done in a different way than in the non-binary case. If a tuple is not valid then one of its values has been removed from the domain of the corresponding variable. This means that the tuple has also be removed from the domain of the hidden variable. Therefore, to check the validity of a tuple we only need to look in the domain of the hidden variable and check if the tuple is present.

We will now show that the GAC algorithm of Figure 2 and its corresponding AC algorithm on the encoding will perform the same number of consistency checks when applied on a problem that is GAC. Consider that if no domain wipeout in any variable (original or hidden) occurs then the two algorithms will add constraints (hidden variables) to the stack and remove them for revision in exactly the same order. The difference is that the binary version will revise domains of hidden variables as an extra step. However, this does not involve any consistency checks. Therefore, we only need to show that if a value is deleted from a variable during the revision of a constraint or finds a new support in the constraint then these operations will require the same number of checks in both representations. Assume that in the non-binary version of the algorithm value a

is deleted from variable x because it has no support in constraint c. If $|T|$ is the number of allowed tuples in c then this will require $|T| - currentSupport_{x,a,c}$ checks, one for each of the tuples in c that have not been checked yet. If the value is not deleted but finds a new support τ, with $\tau > currentSupport_{x,a,c}$, then $\tau - currentSupport_{x,a,c}$ checks will be performed. In the hidden variable encoding, x will be processed in the same order as in the non-binary version and we will require $|T| - currentSupport_{x,a,h_c}$ or $\tau - currentSupport_{x,a,h_c}$ checks depending on the case. h_c represents the hidden variable corresponding to c. Obviously, both supports are the same, since a tuple in c corresponds to a value in h_c, and the same number of checks will be performed in both representations.

On the other hand, on a problem that is not GAC, the AC algorithm on the encoding can perform less checks than the GAC algorithm. Consider a problem that includes variables x_1, x_2, x_3, x_4 with domains $\{0,1\}$, $\{0,1\}$, $\{0,\ldots,9\}$, and $\{0,1\}$, respectively. There are two constraints, c and c', over variables (x_1,x_2,x_3) and (x_1,x_2,x_4) respectively. Value 0 of x_2 is supported in c by tuples that include the variable-value pair $(x_1,1)$. Value 0 of x_1 is supported in c' by tuples that include the variable-value pair $(x_2,0)$. Values $0,\ldots,9$ of x_3 are supported in c by tuples that include $(x_2,0)$ and by tuples that include $(x_2,1)$. Assume that variable x_1 is instantiated to 0, which means that the deletion of 1 from x_1 must be propagated. In the encoding, we will first delete all tuples that include the value $(x_1,1)$ from hidden variables h_c and $h_{c'}$. Then, we revise all original variables connected to hidden variables h_c and $h_{c'}$. Assuming that h_c is processed first, value 0 of x_2 will have no support in h_c so it will be deleted. As a result, we will delete all tuples from hidden variable $h_{c'}$ that include the pair $(x_2,0)$. This means that the domain of $h_{c'}$ will be wiped out. In the non-binary representation, after the deletion of 0 from x_2, we will find that value 1 of x_2 and all values of x_3 have supports in c. This will involve checks that are avoided in the encoding. The inconsistency will be discovered when we process constraint c' and find out that value 1 of x_2 has no support in c' resulting in the domain wipeout of x_2.

We have demonstrated that AC in the hidden variable encoding can detect an inconsistency with fewer checks than GAC in the non-binary representation, while on graphs that are AC both algorithms will perform the same checks. This does not mean that algorithms on the encoding will always be more efficient in run times because the run time of an algorithm depends on the total number of primitive operations it will perform. There is a trade-off in the operations that the GAC algorithm performs in the non-binary version compared to the binary one. Assuming there are k_p past (instantiated) and k_f future variables in a constraint with $|T|$ allowed tuples then the binary GAC-3 algorithm will, in the worst case, perform $O(k_f d^{k_f})$ checks + $O(|T|)$ updates in the domain of the hidden variable, when applied on the encoding. That is, the worst-case complexity in the number of primitive operations is $O(k_f d^{k_f} + |T|)$. The non-binary GAC-3 will perform $O(k k_f d^{k_f})$ operations in the worst case. That is, for every check, the algorithm will have to make $O(k)$ domain checks to make sure that the checked tuple is valid.

4 Search Algorithms

Like GAC algorithms, non-binary search algorithms can be simulated by equivalent algorithms that run on the hidden variable encoding. For example, it has been shown that the MGAC algorithm on a non-binary CSP is equivalent to MAC on the hidden variable encoding of the CSP when only original variables are instantiated and similar branching heuristics are used [12]. We now show that similar results hold for generalized versions of forward checking (FC).

According to the simplest generalization of FC, forward checking is performed only after k-1 variables of an k-ary constraint have been instantiated. This algorithms is called nFC0 in [3]. More, and stronger, generalizations of FC to non-binary constraints were introduced in [3]. These generalizations differ between them in the extent of look-ahead they perform after each variable instantiation. For example, algorithm nFC5, which is the strongest version, tries to make the set of constraints involving at least one past variable and at least one future variable GAC. All the generalizations reduce to simple FC when applied to binary constraints.

Here we will show that the various versions of nFC are equivalent, in terms of visited nodes, to binary versions of FC that run on the hidden variable encoding of the problem. As mentioned, this holds under the assumption that the binary algorithms only instantiate original variables and they use similar branching heuristics as their non-binary counterparts. We call these binary algorithms hFC0–hFC5. Each binary algorithm performs the same amount of propagation as the corresponding non-binary algorithm. For example, hFC5 will enforce AC on the set of hidden variables, and original variables connected to them, such that each hidden variable is connected to at least one past original variable and at least one future original variable. The equivalence between nFC1 and an algorithm called FC+ in [1] has already been proven in [3].

Proposition 1. *In any non-binary CSP, algorithms nFC0–nFC5 are equivalent to binary forward checking algorithms hFC0–hFC5 that operate on the hidden variable encoding of the problem resulting in the same node visits.*

Proof. We prove this for nFC5, the strongest among the generalized FC algorithms. Proofs for the other versions are similar. We only need to prove that at each node of the search tree algorithms nFC5 and hFC5 will delete exactly the same values from original variables. Assume that at some node, after instantiating the current variable, nFC5 deletes value a from a future variable x because it found no support in a constraint c that has at least one instantiated variable. hFC5 will also delete this value from x because it will find no consistent tuple in the corresponding hidden variable h_c. This is due to the fact that the current domain of h_c will contain only valid tuples with respect to the current variable domains of the original variables, since inconsistent ones will have been deleted either in a previous run of AC, or after the instantiation of the current variable (recall that h_c contains at least one instantiated variable). Now in the opposite case, if hFC5 deletes value a from an original variable x it means that all tuples including that assignment are not present in the domains of a hidden variable

h_c that include x and at least one past variable. In other words, there is no consistent tuple in c, with respect to the current variable domains, that contains the assignment $x = a$. As a result, nFC5 will remove a from the domain of x.

□

Therefore, if we never instantiate hidden variables in the binary representation and apply algorithms hFC0–hFC5 we will end up with the same node visits as the respective nFC0–nFC5 algorithms in the non-binary representation. Note that in [1] experimental results show differences between FC on the hidden variable encoding and non-binary FC. However, the algorithms compared there were FC+ and nFC0 which are not equivalent. We have also experimented with a stronger version of hFC5, which we call hFC5b, that visits fewer nodes than nFC5 and hFC5 but may perform more operations at each node. hFC5b is a FC algorithm that operates exactly like hFC5 in that no original variable involved in constraints that contain only future variables is revised. If however a value is deleted from some future variable x because of a constraint between x and past variables then all hidden variables connected to x are revised, including hidden variables that are only connected to future originals. Observe that there is no equivalent to hFC5b that applies on the non-binary representation. In general, the hidden variable encoding is a flexible representation that allows for the definition of algorithms that maintain more refined consistency levels depending on which hidden variables are updated.

5 Instantiating Hidden Variables

So far we have shown that solving an extensionally defined CSP by using the non-binary representation is in many ways equivalent to solving it using the hidden variable encoding, assuming that only original variables are instantiated. A natural question is whether search techniques which are inapplicable in the non-binary case can be applied on the encoding. The answer is the ability of a search algorithm that operates on the encoding to select and instantiate hidden variables. In the equivalent non-binary representation this would imply instantiating values of variables simultaneously. To implement such an algorithm we would have to modify standard search algorithms and heuristics or devise new ones. On the other hand, in the hidden variable encoding an algorithm that instantiates hidden variables can be easily implemented using a standard search algorithm and branching heuristic. Note, that if we only instantiate original variables then the hidden variables will be instantiated implicitly. That is, when all the original variables connected to a hidden are instantiated then the domain of the hidden variable is reduced to a singleton (i.e., it is instantiated). As the next section shows, by instantiating hidden variables in the encoding we can also achieve higher levels of consistency than in the non-binary representation.

5.1 Singleton Consistencies

We know that enforcing AC in the hidden variable encoding is equivalent to enforcing GAC in the original problem. Here we prove that when we move up to the consistency level of SAC then enforcing it on the hidden variable encoding is strictly stronger than enforcing SGAC on the original problem. This is derived from the ability of SAC to istantiate hidden variables and check their consistency. We denote by $P_{D_i=\{a\}}$ the CSP obtained by restricting the domain of variable x_i to $\{a\}$ in a CSP P.

Proposition 2. *Achieving singleton arc consistency on the hidden variable encoding of a non-binary problem is strictly stronger than achieving singleton generalized arc consistency on the variables in the original problem.*

Proof. We have to prove that if a value a of a variable x_i in a CSP P is not SGAC then SAC on the encoding of P will prune that value. From [12] we know that if a value b of variable x_j is not GAC in $P|_{D_i=\{a\}}$ then it is also arc inconsistent in the encoding of $P|_{D_i=\{a\}}$. For SGAC to remove value a, all values in a variable x_j must be deleted when a is assigned to x_i. According to the above, all such values will also be deleted from the domain of x_i in the hidden variable encoding of $P|_{D_i=\{a\}}$. Therefore, value a will be singleton arc inconsistent in the hidden variable encoding. To show strictness, consider a problem with five variables$\{x_1, x_2, x_3, x_4, x_5\}$, all of them with domain $\{0, 1\}$, and the following ternary constraints: A constraint over $\{x_1, x_2, x_3\}$ with allowed tuples $\{< 0, 0, 1 >, < 0, 1, 0 >, < 1, 0, 0 >, < 1, 1, 1 >\}$, a constraint over $\{x_1, x_2, x_4\}$ with allowed tuples $\{< 0, 0, 1 >, < 0, 1, 0 >, < 1, 0, 0 >, < 1, 1, 1 >\}$, and a constraint over $\{x_1, x_2, x_5\}$ with allowed tuples $\{< 0, 1, 0 >, < 1, 0, 1 >\}$, Enforcing SGAC on this problem will make no deletions. However, enforcing SAC on the encoding will show that the problem is insoluble. If we take the hidden variable h_1 corresponding to the constraint over $\{x_1, x_2, x_3\}$, for example, enforcing SAC will delete all the tuples from its domain because they are all singleton arc inconsistent.

□

In [12] it is proved that all consistency levels between SAC and AC (e.g. path inverse consistency and restricted path consistency) collapse onto AC, in the hidden variable encoding. Also, neighborhood inverse consistency, which is incomparable to SAC collapses onto AC. Therefore, the weakest consistency level where we notice a gap between the amount of pruning achieved in the hidden encoding and the non-binary representation is SAC. In fact, to get the pruning achieved by SAC in the encoding we only need to consider the hidden variables. For example, if all tuples in a hidden variable that include the variable-value pair (x, a) are removed by SAC then so will the value a from x. However, the extra pruning achieved in the encoding incurs extra cost because of the (usually) large domain sizes of the hidden variables. If we restrict SAC on encoding to the original variables only then we get the same level of consistency as SGAC in the original problem. The proof is easy and is omitted due to space restrictions.

6 Experimental Results

In this section we study empirically the efficiency of algorithms that run on the hidden variable encoding compared to their non-binary counterparts. For the empirical investigation we use randomly generated problems and benchmark crossword puzzle generation problems. Both of these classes are naturally defined by an extensional representation of the constraints. In the case of crossword puzzles the constraints are by nature very tight. In the case of random problems we also focus our attention on tight instances. The reason being that the binary encoding can only be practical if the constraints are tight enough so that the domains of the hidden variables are not prohibitively large.

6.1 Random Problems

Random problems were generated using the extended model B as in [3]. Under this model, a random CSP is defined by five parameters $< n, d, k, p, q >$, where n is the number of variables, d the domain size, k the arity of the constraints, p the density of the generated graph, and q the looseness of the constraints. p and q are given as a % percentage of the constrained variable combinations and allowed tuples in these constraints, respectively. In this empirical comparison we included the following algorithms: MGAC, MHAC, which stands for MAC in the encoding that only instantiates original variables, nFC5, hFC5, and hFC5b. hFC5 and hFC5b also instantiate only original variables. All algorithms use the dom/deg heuristic for variable ordering [4] and lexicographic value ordering. The GAC and AC algorithms used are the ones described in Sections 3.1 and 3.2. We chose to use these algorithms because they have a good asymptotic complexity and they are easy to implement. We do not include results on algorithms that can instantiate hidden variables as well as original ones because experiments showed that such algorithms have very similar behavior to the corresponding algorithms that instantiate only original variables. The reason is that, because of the nature of the constraints, the dom/deg heuristic almost always selects original variables. In the rare cases where the heuristic selected hidden variables, this resulted in an increase in node visits. Table 1 shows the performance of the algorithms on four classes of randomly generated ternary CSPs. All classes are from the hard phase transition region. Classes 1 and 2 are sparse, 3 is very sparse, and 4 is again relatively sparse but denser than the others. We report node visits, CPU times, and consistency checks. A consistency check consists of two operations. 1) Checking if a tuple τ includes the value for which we search for support, and 2) checking if τ is valid.

From Table 1 we can see that algorithms that operate on the encoding and instantiate only original variables perform fewer checks in all classes than the corresponding non-binary algorithms. This is due to their ability of early domain wipeout detection at dead ends. CPU times are influenced not only by the number of checks but by the total number of primitive operations performed. We can see that MHAC performs better than MGAC on the sparser problems. However, the differences in classes 1 and 2 are marginal. In general, for all the 3-ary classes we tried with density less than $3\% - 4\%$ the relative performance of MHAC and MGAC (in run times) ranged from being equal to a 40% advantage for MHAC. The differences are more notable on the very sparse class 3. This is due to the fact that for sparse problems the hard region is located at

	nFC5	hFC5	hFC5b	MGAC	MHAC
class 1: $n = 30$, $d = 6$, $k = 3$, $p = 1.847$, $q = 50$					
nodes	4645	4645	4150	3430	3430
sec	1.47	1.65	1.90	2.08	1.90
checks	13M	11M	10M	20M	14M
class 2: $n = 75$, $d = 5$, $k = 3$, $p = 0.177$, $q = 41$					
nodes	21976	21976	16723	7501	7501
sec	5.67	6.90	5.63	4.09	3.41
checks	17M	16M	12M	24M	15M
class 3: $n = 50$, $d = 10$, $k = 5$, $p = 0.001$, $q = 0.5$					
nodes	21283	21283	20260	16496	16496
sec	58.56	22.25	27.73	74.72	22.53
checks	783M	643M	631M	847M	628M
class 4: $n = 20$, $d = 10$, $k = 3$, $p = 5$, $q = 40$					
nodes	5400	5400	5124	4834	4834
sec	4.19	5.19	7.78	5.75	8.15
checks	119M	99M	95M	151M	119M

Table 1. Comparison of algorithms on sparse random classes. Classes 1 and 2 taken from [3]. CPU times are in seconds. For nodes and checks we give mean numbers for 50 instances at each class. "K" implies $\times 10^3$ and "M" implies $\times 10^6$

low constraint tightnesses (i.e., small domains for hidden variables) where only a few operations are required for the revision of hidden variables. Another factor contributing to the dominance of the binary algorithms in class 5 is the arity of the constraints. The non-binary algorithms require more operations to check the validity of tuples when the tuples are of large arity, as explained in Section 3.1.

When the density of the graph increases (class 4), the overhead of revising the large domains of hidden variables and restoring them after failed instantiations slows down the binary algorithms, and as a result they are outperformed by the non-binary ones. For denser classes than the ones reported, the phase transition region is at a point where more than half of the tuples are allowed, and in such cases the non-binary algorithms perform even better.

6.2 Crossword Puzzles

Crossword puzzle generation problems have been used for the evaluation of search heuristics for CSPs [9, 2] and binary encodings of non-binary problems [1, 12]. Tables 2 and 3 show the performance of the tested algorithms for various crossword puzzles in running time and number of visited nodes. We used selected hard puzzles from [9] and 20 15×15 and 19×19 puzzles from [2]. Apart from algorithms that instantiate only original variable we tested versions of hFC5 and MAC which may also instantiate hidden variables. We call these algorithms hidFC5, hidFC5b, and hidMAC. Again, all algorithms use the dom/deg heuristic for variable ordering. An em-dash (—) is placed wherever some method did not manage to find a solution within 5 hours of cpu-time. **n** is the number of words and **m** is the number of blanks in each puzzle. Problems marked by (*) are insoluble.

We used the Unix dictionary for the allowed words in the puzzles. Four puzzles (15.06, 15.10, 19.03, 19.04) could not be solved by any of the algorithms within 5 hours of cpu time. Also two puzzles (19.05 and 19.10) were arc inconsistent. GAC discovered the inconsistency slower than HAC in both cases (around 3:1 time difference in 19.05 and 10:1 in 19.10) because the latter method discovered early the domain wipe-out of a hidden variable.

At the rest of the puzzles we can observe that MHAC usually performs better than MGAC on the hard instances. For the hard insoluble puzzles the difference is considerable, and so is the difference between hFC5 and nFC5. This is mainly due to the uniformly large arity of the constraints in these classes.[4] Another interesting observation is that there can be large differences between the performance of methods that instantiate hidden variables and those which instantiate only original ones. In many cases hidMAC managed to find a (different) solution than MHAC and MGAC earlier. This shows that we can benefit from a method that instantiates hidden variables. In puzzle 19.08 hidMAC managed to find a solution fast, while the other MAC algorithms thrashed. Note, that the FC algorithms also found a solution quickly, which means that in this case the propagation of MGAC and MHAC misguided the variable ordering heuristic. On the other hand, the hid* methods were also subject to thrashing in instances where other methods terminate. The fact that in all insoluble puzzles hidMAC did not do better than MHAC shows that its performance is largely due to the variable ordering scheme. When comparing MAC methods with equivalent FC5 ones, we see that in most cases maintaining full consistency is better for this class of problems. Also, the hFC5b and hidFC5b algorithms do not always pay-off.

Regarding node visits, observe that in many cases hidden variable instantiation methods visit less nodes than their original variable counterparts, but this does not reflect to the same time performance difference because when a hidden variable is instantiated hidMAC does more work than when an original one is. It has to instantiate automatically all original variables involved in the hidden and propagate these changes to all other hidden variables containing them. Note, that constraints in crosswords are much tighter than the constraints in random problems. For example, the tightness of a 6-ary constraint in a puzzle is 99,999988%. This is why the hid* methods can perform well on such problems. Consistent problems with such high tightnesses cannot be generated randomly.

In general, we believe that if we exploit better the potential of instantiating hidden variables (i.e., by a suitable variable ordering heuristic), methods that instantiate hidden variables can go down the search tree faster than ones that consider only original variables, because they can benefit from small hidden variable domains. Notice that hidMAC reduces to MHAC if it instantiates only original variables. Therefore, if employed with the optimal variable ordering it can never be worse than MHAC. We are currently working towards devising such ordering heuristics.

[4] Puzzles 6×6–10×10 correspond to square grids with no blank squares.

puzzle	n	m	MGAC	MHAC	hidMAC	nFC5	hFC5	hidFC5	hFC5b	hidFC5b
15.01	78	189	8.5	7.9	4.4	11.5	15.4	5.3	10.1	4.2
15.02	80	191	24.5	26.9	—	77.8	138.7	—	61.1	—
15.03	78	189	4.2	4.6	2.3	21.2	30.6	2.3	30.9	2.81
15.04*	76	193	290	295	218	24.5	29.8	979	243	791
15.05	78	181	3	3.1	2.2	3.7	3.8	3.3	4.8	2.5
15.07	74	193	670	335	376m	48.3	39.4	482m	465m	367m
15.08	84	186	2.32	2.27	2.89	3.22	3.37	3.52	3.27	3.1
15.09	82	187	2.24	2.3	2.45	1.92	1.81	—	2.43	—
19.01	128	301	7.6	7.3	6.9	—	—	4.56	—	4.8
19.02	118	296	198	204	—	—	—	—	495	—
19.06	128	287	5.9	4.7	5.8	4.1	4.9	4.6	5	—
19.07	134	291	3.4	3.4	4.4	4.1	4.1	5.2	3.8	5.2
19.08	130	295	—	—	5.45	4	3.3	4.7	3.6	4.7
19.09	130	295	3.64	5	4.2	6.2	6.7	4.6	4.8	4.8
puzzleC	78	189	77.5	107	—	153	209	—	115	—
6×6	12	36	84	55	64	109	75	104	73	79
7×7*	14	49	120m	75m	96m	176m	107m	159m	120m	148m
8×8*	16	64	45m	29m	42m	58m	32m	57m	35m	59
9×9*	18	81	488	337	454	868	470	737	614	797
10×10*	20	100	117.7	77	93	534	331	363	192	217

Table 2. Comparison (in cpu time) of algorithms on crossword puzzles. All times are in seconds except those followed by "m" (minutes).

7 Conclusion

In this paper, we performed a theoretical and empirical investigation of arc consistency and search algorithms for the hidden variable encoding of non-binary CSPs. We analyzed the potential benefits of using AC algorithms on the hidden encoding compared to GAC algorithms on the non-binary representation. We showed that FC algorithms for non-binary constraints can be emulated by corresponding binary algorithms that operate on the hidden variable encoding and only instantiate original variables. Empirical results on various implementations of search algorithms showed that the hidden variable is competitive and in many cases better than the non-binary representation for tight classes of non-binary constraints. A general conclusion from this study is that there is an interesting mapping between algorithms for non-binary constraints and corresponding algorithms for binary encodings, even in refined levels of implementation. For future work we plan to develop variable ordering heuristics more suitable to the hidden encoding. Also, we intend to investigate how lessons learned from this study apply to other GAC algorithms, like GAC-schema.

Acknowledgements

The second author is a member of the APES research group and would like to thank all other members. Especially, Peter van Beek, Ian Gent, Patrick Prosser and Toby Walsh. We would also like to thank Christian Bessière.

puzzle	n	m	MGAC,MHAC	hidMAC	nFC5,hFC5	hidFC5	hFC5b	hidFC5b
15.01	78	189	574	200	1607	398	1067	295
15.02	80	191	1312	—	15559	—	6029	—
15.03	78	189	338	126	4105	159	3364	183
15.04*	76	193	19667	18479	2869	75450	25202	63985
15.05	78	181	286	145	528	248	459	189
15.07	74	193	12733	568768	4180	1504450	2700150	744180
15.08	84	186	247	165	362	277	294	187
15.09	82	187	251	155	247	—	287	—
19.01	128	301	469	309	—	224	—	202
19.02	118	296	15764	—	—	—	33079	—
19.06	128	287	375	158	357	200	346	—
19.07	134	291	305	206	344	240	306	222
19.08	130	295	—	191	332	249	322	218
19.09	130	295	308	167	458	199	347	171
puzzleC	78	189	9827	—	26315	—	11820	—
6×6	12	36	2263	2097	7332	5735	5028	4259
7×7*	14	49	116082	138199	634858	455716	396791	303330
8×8*	16	64	31386	40037	231950	163527	108338	78076
9×9*	18	81	4972	5715	71020	35736	23279	14344
10×10*	20	100	1027	1120	35492	18922	13105	10438

Table 3. Comparison (in node visits) of algorithms on crossword puzzles. MGAC and MHAC visit the same number of nodes and this holds also for nFC5 and hFC5.

References

1. F. Bacchus and P. van Beek. On the Conversion between Non-Binary and Binary Constraint Satisfaction Problems. In *Proceedings of AAAI'98*, pages 310–318, 1998.
2. A. Beacham, X. Chen, J. Sillito and P. van Beek. Constraint programming lessons learned from crossword puzzles. In *Proceedings of the 14th Canadian AI Conf.*, 2001.
3. C. Bessière, P. Meseguer, E.C. Freuder, and J. Larrosa. On Forward Checking for Non-binary Constraint Satisfaction. In *Proceedings of CP'99*, pages 88–102, 1999.
4. C. Bessière and J.C. Régin. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In *Proceedings of CP'96*, pages 61–75, 1996.
5. C. Bessière and J.C. Régin. Arc Consistency for General Constraint Networks: Preliminary Results. In *Proceedings of IJCAI'97*, pages 398–404, 1997.
6. C. Bessière and J.C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'2001.*
7. X. Chen. A Theoretical Comparison of Selected CSP Solving and Modeling Techniques. PhD thesis, University of Alberta, Canada, 2000.
8. R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI'97*, pages 412–417, 1997.
9. M. Ginsberg, M. Frank, M. Halpin, and M. Torrance. Search lessons learned from crossword puzzles. In *Proceedings of AAAI-90*, pages 210–215, 1990.
10. R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of ECAI'88*, pages 651–656, 1988.
11. F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings of ECAI'90*, pages 550–556, 1990.
12. K. Stergiou and T. Walsh. Encodings of Non-Binary Constraint Satisfaction Problems. In *Proceedings of AAAI'99*, pages 163–168, 1999.
13. K. Stergiou and T. Walsh. On the complexity of arc consistency in the hidden variable encoding of non-binary CSPs. *Submitted for publication.*
14. Y. Zhang and R. Yap. Making AC-3 an optimal algorithm. In *Proceedings of IJCAI'2001.*