The Dilemma Between Arc and Bounds Consistency

Nikolaos Pothitos and Panagiotis Stamatopoulos

Department of Informatics and Telecommunications
National and Kapodistrian University of Athens
Panepistimiopolis, 157 84 Athens, Greece
{pothitos,takis}@di.uoa.gr

Abstract

Consistency enforcement is used to prune the search tree of a Constraint Satisfaction Problem (CSP). Arc Consistency (AC) is a well-studied consistency level, with many implementations. Bounds Consistency (BC), a looser consistency level, is known to have equal time complexity to AC. To solve a CSP, we have to implement an algorithm of our own or employ an existing solver. In any case, at some point, we have to decide between enforcing either AC or BC. As the choice between AC or BC is more or less predefined and currently made without considering the individualities of each CSP, this work attempts to make this decision deterministic and efficient, without the need of trial and error. We find that BC fits better while solving a CSP with its maximum domains' size being greater than its constrained variables number. We study the behavior of *Maintaining* Arc or Bounds Consistency during search, and we show how the *overall* search methods complexity is affected by the employed consistency level.

Keywords: propagation, search, MAC, bounds consistency, constraint satisfaction

1 Introduction

Constraint Programming (CP) aims at solving *Constraint Satisfaction Problems* (CSPs) in a transparent way: the user simply states the problem and the computer solves it.[1,2] The consequence of this "motto" is that the solver should decide automatically on its own which algorithm will solve a given CSP without human intervention; the role of the user is limited just to *define* the CSP.

This elegant separation of the user experience and the internal solving process is what makes Constraint Programming an intelligent paradigm, and this is the motivation behind this work. We focus on an essential part of the solving process called *consistency enforcement* and develop criteria that help Constraint Programming solvers select the fastest between arc consistency (AC) and bounds consistency (BC), without human intervention.

1.1 Constraint Satisfaction Problems

CSPs cover a wide range of problems, including planning and scheduling,[4] logic puzzles,[14] all Boolean satisfiability problems,[19] circuit design,[24] robotics,[17] and many others. CSPs are widespread because they express many problems that occur in real life.

Definition 1. A binary *Constraint Satisfaction Problem* (CSP) consists of

- a set of constrained *variables* $\mathscr{X} = \{X_1, X_2, \ldots, X_n\}$,
- the corresponding set of *domains* $\mathscr{D} = \{D_1, D_2, \ldots, D_n\}$ which are finite sets (of integer values in this work) and
- the set of *constraints* between the variables $\mathscr{C} = \{C_{ij} \,|\, X_i, X_j \in \mathscr{X}, i \neq j\}$.
  - If $i < j$, then $C_{ij} = (i, j, R_{ij})$ with $R_{ij} \subseteq D_i \times D_j$. An assignment of a value v_i to X_i and v_j to X_j is valid with respect to C_{ij}, if and only if $(v_i, v_j) \in R_{ij}$.
  - If $i > j$, then C_{ij} is equivalent to C_{ji}, in the sense that if an assignment of a value v_i to X_i and v_j to X_j is accepted by C_{ji}, then it is accepted by C_{ij} too.

If we assign a value to every variable, and the assignments are valid with respect to every constraint, then the assignment is a *solution*.

As implied in the above definition, in this work we consider only *binary* CSPs, plainly for the sake of simplicity. Each constraint of a binary CSP affects no more than *two* variables. However, our scope is not limited to binary CSPs, because it has been proved that any non-binary CSP can be transformed into an equivalent binary one.[25]

We do not also refer to unary constraints in this work, due to their triviality. A unary constraint applies onto a single variable. The initial domain of each variable should be shrunk to include only the values permitted by the corresponding unary constraint. This action can be performed as a single preprocessing step, before proceeding to actually solve a CSP.

1.2 The Intelligence behind Constraint Propagation

Let us say that we have to crack a password $\langle X_1, X_2, X_3 \rangle$ consisting of three decimal digits. In order to guess them, we are given hints for five combinations.

1	$\langle 6, 8, 2 \rangle$	One number is correct and well placed
2	$\langle 6, 1, 4 \rangle$	One number is correct but wrongly placed
3	$\langle 2, 0, 6 \rangle$	Two numbers are correct but wrongly placed
4	$\langle 7, 3, 8 \rangle$	Nothing is correct
5	$\langle 7, 8, 0 \rangle$	One number is correct but wrongly placed

We can naturally model this puzzle as a constraint satisfaction problem of three variables X_1, X_2, X_3, with initial domains $D_1 = D_2 = D_3 = \{0, 1, 2, \ldots, 9\}$.

To solve this puzzle, it is *not* necessary to iterate through all the 1,000 candidate passwords and check them against the given constraints. An intelligent method would *propagate* the constraints of the above table.

- From the 4^{th} constraint, we conclude that the domains of the variables do not contain 7, 3, and 8.
- From the above and the 5^{th} constraint, we conclude that $X_1 = 0$ or $X_2 = 0$. After all, the values 7 and 8 are incorrect, so 0 is correct, but wrongly placed.
- From the above and the 3^{rd} constraint, we conclude that $X_1 = 0$, as we are told that 0 is wrongly placed as the second digit.
- From the first two constraints, 6 cannot be a correct digit.
- Therefore, from the 1^{st} constraint, $X_3 = 2$, as neither 6 nor 8 can be correct digits.
- Finally, from the 2^{nd} constraint, $X_2 = 4$, because if 1 was correct, we should assign it either to X_1 or to X_3, which would contradict the above.

This was a *constraint propagation* example that directly gave the solution $\langle X_1, X_2, X_3 \rangle = \langle 0, 4, 2 \rangle$. Normally, in other CSPs, constraint propagation should be combined with a search method. But in any case, it is apparent that propagation can dramatically reduce the search space, i.e. the set of candidate solutions that we should check.

Constraint propagation is a form of inference [5] and reasoning, [6] and, as such, it is an intelligent methodology incorporated in intelligent constraint programming solvers. Consistency enforcement is a formalized way of constraint propagation.

1.3 Consistency Enforcement

Consistency is a very useful property in the road to solve a CSP. It implies that the values of the domains of each variable have a kind of *support* with respect to the CSP constraints.

Definition 2. An arc (X_i, X_j) is *arc consistent* iff for each $v_i \in D_i$ there exists a $v_j \in D_j$ with (v_i, v_j) not violating C_{ij}.

Example 1. Let X_1 and X_2 be two constrained variables with domains $D_1 = \{1, 2, 3\}$ and $D_2 = \{2, 3, 4, 5, 6, 7\}$. Let us assume that the constraint between the variables is $X_2 = 2 \cdot X_1$.

(X_1, X_2) is arc consistent, as for each of the values 1, 2, 3 in D_1, the corresponding values 2, 4, 6 belong to D_2.

On the other hand, (X_2, X_1) is *not* arc consistent. To prove this, we need just one value from D_2 that does not have any support in D_1. Indeed, for the value 3 in D_2, there is not any v_1 in D_1 with $2 \cdot v_1 = 3$.

If we want to make (X_2, X_1) arc consistent, we should remove the values 3, 5, 7 out of D_2 as they do not have any supports in D_1.

This example also illustrates that consistency is not a symmetric property.

In order to check if an arc (X_i, X_j) is arc consistent, we have to iterate through all the values of D_i. The function that does this and removes the unsupported values from D_i is called Revise. A faster yet looser alternative would be to check if the arc is *bounds* consistent.

Definition 3. An arc (X_i, X_j) is *bounds consistent* iff for the $\min D_i$ and $\max D_i$ values, there exist some v_a, $v_b \in D_j$ with $(\min D_i, v_a)$ and $(\max D_i, v_b)$ not violating C_{ij}.

Formally, Definition 3 is about the so-called *bounds(D)* consistency and not the bounds(Z) and bounds(R) variants.[5]

In this case, REVISE has to check and update only the two bounds of D_i. But, in the worst case, when no support is found, it has to iterate through all D_i values too.

Example 2. Again, let X_1 and X_2 be two variables with $D_1 = \{1, 2, 3\}$, $D_2 = \{2, 3, 4, 5, 6, 7\}$, and $X_2 = 2X_1$.

(X_1, X_2) is bounds consistent, as for each of the bounds 1 and 3 in D_1, the corresponding values $2 \cdot 1 = 2$ and $2 \cdot 3 = 6$ belong to D_2.

Nevertheless, (X_2, X_1) is bounds inconsistent, as the upper bound 7 of D_2 has not any support in D_1.

If we want to enforce bounds consistency to (X_2, X_1), we should remove 7 out of D_2. Note that only one removal is needed in the case of bounds consistency enforcement in contrast to the three removals needed in the arc consistency enforcement for the same domains in Example 1.

Lemma 1. *Both arc and bounds consistency enforcement have equal time complexities in the worst case.*

Proof. Time is measured by counting the number of elementary steps that each algorithm takes. We use the common *uniform unit system* in which every algorithm's operation takes the same constant time.[30]

In order to compute the worst-case complexity of enforcing arc consistency, we consider the following procedure.

```
1  function ReviseAC(X_i, X_j)
2      for each v_i ∈ D_i do
3          value_is_supported ← false
4          for each v_j ∈ D_j do
5              if (v_i, v_j) ∈ R_ij, with C_ij ∈ 𝒞 then
6                  value_is_supported ← true
7                  break
8              end if
9          end for
10         if value_is_supported then
11             continue
12         else
13             Remove v_i out of D_i
14         end if
15     end for
16 end function
```

Let d be the maximum domain size. Then, line 2 performs at most d iterations. Line 3 is 1 elementary operation. The loop in line 4 performs at most d iterations. The statements inside this inner loop are at most 3 elementary operations. Finally, lines 10–14 consist at most 4 elementary operations.

Overall, we have at most $d \cdot (1 + d \cdot 3 + 4)$ elementary operations, which is $O(d^2)$.

Bounds consistency enforcement is a variation of the above.

```
function ReviseBC(X_i, X_j)
    for each v_i ∈ D_i in ascending order do
        value_is_supported ← false
        for each v_j ∈ D_j do
            if (v_i, v_j) ∈ R_ij, with C_ij ∈ 𝒞 then
                value_is_supported ← true
                break
            end if
        end for
        if value_is_supported then
            break
        else
            Remove v_i out of D_i
        end if
    end for
    for each v_i ∈ D_i in descending order, with v_i > min D_i do
        value_is_supported ← false
        for each v_j ∈ D_j do
            if (v_i, v_j) ∈ R_ij, with C_ij ∈ 𝒞 then
                value_is_supported ← true
                break
            end if
        end for
        if value_is_supported then
            break
        else
            Remove v_i out of D_i
        end if
    end for
end function
```

Typically, ReviseBC is similar to ReviseAC, but contains two loops instead of one. The number of elementary steps inside each loop is still at most $(1 + d \cdot 3 + 4)$.

The number of iterations of the first loop plus the number of iterations of the second loop is at most d, because, in the worst case, the algorithm iterates through all the values of D_i. Each respective value of D_i is visited at most once.

Overall, similarly to ReviseAC, the number of elementary operations is again $d \cdot (1 + d \cdot 3 + 4)$ which is $O(d^2)$. □

To put it straight, enforcing arc or bounds consistency between a pair of constrained variables (X_i, X_j) takes the same time if X_i has not any support in X_j, which results in removing every value out of D_i. This is the worst case.

Nevertheless, in a better case, if ReviseBC finds a support, it stops the corresponding iteration through D_i values, while ReviseAC always iterates through all of them.

1.4 Our Contribution and Alternative Approaches

From Constraint Programming early years, developers of solvers such as Ilog have observed empirically that there is a trade-off between arc and bounds

consistency in terms of time and space, and bounds consistency is preferable in many cases. [27]

In alternative approaches to this work, in current constraint programming solvers, the choice between AC and BC is not justified theoretically but only empirically. In our work, apart from wide experimental results, we provide theoretical analysis for the AC vs. BC trade-off so as to predict when arc consistency becomes a bottleneck. We show that bounds consistency is usually more efficient when dealing with CSPs having large domains.

This could be thought of as a paradox, because AC and BC have equal worst-case complexities, and AC is stronger than BC, in the sense that it removes more inconsistent values out of the domains of constrained variables. This is true, but only when we study the constraint propagation algorithms isolated, independently of the search methods. In this work, we try to see the big picture: constraint propagation integrated into backtracking search methods. We compute the *overall* time complexity and focus on how it is affected by the choice between AC and BC.

1.5 More Related Work

There are various Constraint Programming methodologies and areas. Each different area has been created to serve a different category of CSPs.

1.5.1 Complete and Incomplete Search

Local search and, more specifically, *large neighborhood search* has been integrated into Constraint Programming to solve difficult CSP instances. [20] In such CSPs, we are happy just to find a solution, without usually caring if all candidate solutions will be examined.

Nevertheless, at the beginning, given a specific CSP, one would normally like to make sure if it has any solution or not. This information will be available only by using a standard *backtracking* search method, elaborated in the next Section 2. As, in the worst case, these methods may exhaust all the candidate solutions of a CSP, it is important to make them more intelligent and prune the search space.

1.5.2 Learning from Mistakes or Preventing Them?

Look back techniques in backtracking search methods aim to avoid repeating the invalid assignments of the past. *Backjumping* is a well-known look back technique, but it is not used in solvers, as other techniques clearly outperform it even in simple CSPs. [1] *Nogood learning* is a more promising look back technique that, based on the invalid assignments of the past, adds new constraints to avoid the invalid combination of assignments in the future. [28] However, these new constraints have the drawback of making the constraint network increasingly complex.

On the other hand, *look ahead* techniques are more proactive in the sense that they remove values out of the domains of the constrained variables *before* reaching an inconsistent assignment. *Maintaining arc consistency* (MAC) during search is the queen of all look ahead techniques. [6] According to MAC, each

assignment to a domain of a variable is followed by an arc consistency enforcement method, such as the known optimal AC-2001 algorithm.[7] The optimality of AC-2001 was proven for enforcing arc consistency after a single assignment. But when we call repeatedly AC-2001 during search, after each single assignment, in order to *maintain* arc consistency, there is still room for improvements.[16]

1.5.3 The Importance of Arc Consistency

Arc consistency also plays a key role in splitting the CSPs into two large categories.[6]

1. The *tractable* ones that can be solved in polynomial time, simply by maintaining arc consistency.

2. The *intractable* ones that are NP-complete problems and require an exponential backtracking algorithm to prove whether they have a solution or not.

Related work has defined the properties of the constraint network that suffice to categorize a CSP as tractable or intractable.[10] Furthermore, it has been recently proven that a CSP is tractable only if it contains specific types of constraints.[8,33]

In our work, for the sake of simplicity, we consider arc consistency only for binary constraints. The extension of arc consistency for constraints involving more than two variables is called *generalized arc consistency* (GAC). Contrary to conventional wisdom, there are studies that we can transform non-binary constraints into binary ones and enforce plain AC to them without losing the efficiency of GAC.[29]

1.5.4 Higher-Level Consistencies

As illustrated in Section 1.3 and the included examples, arc consistency is stronger than bounds consistency in the sense that it filters a greater number of futile values out of the domains of the constrained variables. But there are even stronger consistency levels than arc consistency.

These are the so-called *higher-level consistencies* (HLCs) and, while AC examines one constraint at a time, HLCs consider two or more constraints simultaneously. This makes them too expensive to be used in practice.[3] To mitigate the HLC overhead, there are hybrid strategies that go back and forth from HLC to AC.[32] Even machine learning has been employed to dynamically choose which consistency level is more efficient.[2]

1.5.5 Toward More Relaxed Consistencies

In this work, we do not change consistency levels on the fly. We stick to one consistency level at a time (AC or BC) in order to keep the overall search algorithm that maintains consistency as simple as possible. This enables us to shed a more theoretical light to the integration of consistency into search and study the overall consistency complexities, not isolated but always *in the context of search methods* that maintain them. Our computations are backed by wide experimental data.

Instead of swapping HLCs and AC, we choose AC and BC, as bounds consistency is naturally used to describe constraints in Constraint Programming solvers.[15]

In a previous work,[23] instead of switching between different consistency types, consistency was enforced not to all (n) constrained variables but to a varying (k) number of them. However, there was not developed a criterion to guess the best k number of variables a priori.

In Section 2 we present the backbone of constructive search and the related mathematical notation. In Section 3 we compute the upper bounds of the complexities of search methods that traverse a path and maintain either AC or BC. In Section 4 we check in practice if the theoretically computed complexities can predict which methodology, AC or BC, fits better a given CSP.

2 Constructive Search

A typical *backtracking/constructive search method* iterates through the constrained variables of a CSP: it assigns to the first variable a value and proceeds to the second variable, it assigns a value to it and, if the constraints are not violated, proceeds to the third variable and so on. *Backtracking* occurs if any of the constraints is violated: the current assignment is undone, and a different value is assigned to the variable. If all alternative values from the variable's domain are exhausted, we go to the previous variable and assign a different value to it and so on.

2.1 The Standard Backtracking Search Method

Figure 1 illustrates the recursive search method DFS (Depth First Search). Each DFS(ℓ) call corresponds to the variable X_ℓ. In order to solve a CSP, we call DFS(1), to begin with instantiating the first variable X_1. This call attempts to assign to X_1 a value from D_1; hence, we may have at most d different attempts to assign a value to X_1, where d is the maximum size of all the domains. Therefore, we have at most d subsequent calls of DFS(2). Each DFS(2) calls DFS(3) and so on.

This algorithm forms a *search tree*, as in Figure 2. The indicative CSP used in this figure contains three variables X_1, X_2, X_3, with the corresponding domains $D_1 = D_2 = D_3 = \{1, 2\}$. Each level ℓ of the tree refers to a DFS(ℓ) call, and each node of the same level represents an iteration of its **for** loop. More specifically, each node is labeled with the assignment done in line 4.

We have at most d^n leaves representing the lowest level DFS(n) calls, where n is the number of the constrained variables.

Apart from DFS, there are many other backtracking constructive search methods.[22] In any case, DFS is the basis to describe most of them.

2.2 A Search Tree Path

We denote as T_{path} the total time spent in the nodes that belong to the same path. A path begins from the root node and descends to a leaf node. The dotted line in Figure 2 is a path.

```
1  function DFS(ℓ)
       ▷ The method reached the search tree level ℓ:
2      D'_ℓ ← D_ℓ
3      for each v ∈ D'_ℓ do
4          D_ℓ ← {v}      ▷ Assign v to X_ℓ
5          if no constraint is violated then
           ▷ Proceed to the next variable/level:
6              if ℓ = n then
7                  return success
8              else if DFS(ℓ + 1) = success then
9                  return success
10             end if
11         end if
12     end for
13     D_ℓ ← D'_ℓ
14     return failure
15 end function
```

Figure 1: A typical search method

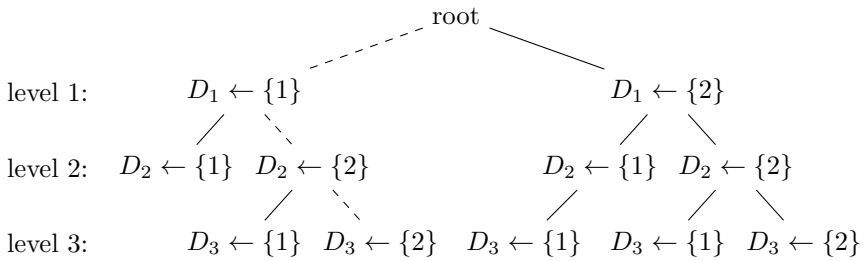

Figure 2: An incomplete binary search tree

$T_{\text{path}}(\ell)$ is a part of T_{path} and denotes the time spent in a node of level ℓ while traversing a path.

In the rest of the paper, the "AC" or "BC" exponents in the above symbols refer to the corresponding AC or BC methodology. For example, $T^{\text{AC}}_{\text{path}}(\ell)$ is the time spent in a node of level ℓ while maintaining AC.

2.3 Paths vs. Trees

Throughout the rest of the theoretic part of this paper, we measure the time spent in search tree *paths*, instead of focusing on the time spent while traversing all the paths of a complete search tree. This is done on purpose, just to simplify our computations.

After all, as it will be proved in the last theoretic section 3.4, if we manage to bound the time needed to traverse a search tree path, we are able to bound the time needed to traverse the whole search tree.

Therefore, we are going to compute respectively an upper bound for traversing a search tree path while maintaining AC or BC, and then multiply it by the maximum number of paths to get an upper bound for the whole search tree.

3 Maintaining Consistency during Search

Depth-first-search method complexity is exponential; we cannot actually decrease its complexity class, but it is possible to limit the number of nodes. In other words, we have to *prune* the tree to make search more efficient, and this can be done via enforcing and maintaining consistency.

3.1 Time Complexity in a Search Tree Node

Figure 3 illustrates a search method with an integrated consistency algorithm that can maintain either arc or bounds consistency. We break up the time spent by DFS_CONS(ℓ) when it is on the top of the call stack into four crucial parts.

- $T_{\text{prop}}(\ell)$ refers to the propagation algorithm in lines 2–4 and 8–14 respectively.
- $T_{\text{store}}(\ell)$ corresponds to line 5 of the algorithm and represents the time needed to store all the initial states of the domains.
- $T_{\text{restore}}(\ell)$ corresponds to line 22 and represents the time needed to restore all the domains. We claim that the time it takes to store the domains is equal to the time it takes to restore them, i.e. $T_{\text{store}} = T_{\text{restore}}$.

  After all, storing the value of a variable requires transferring a specific number of bytes from one place of the memory to another. Re-storing the value back to the variable (the original place of memory) involves the same number of bytes and, therefore, the same number of operations to transfer them back.
- T_{const} corresponds to lines 7 and 15–21. These statements take constant time.

```
1  function DFS_CONS(ℓ)
       ▷ Initially, enqueue all arcs and make them consistent:
2      if ℓ = 1 then
3          CONS    ▷ See Figure 4
4      end if
       ▷ Store a copy of the domains in 𝒟 for a future backtrack:
5      {D'_1, ..., D'_n} ← {D_1, ..., D_n}
6      for each v ∈ D'_ℓ do
7          D_ℓ ← {v}
           ▷ Only the arcs toward X_ℓ are enqueued:
8          Q ← {(X_i, X_ℓ) | C_iℓ ∈ 𝒞}
9          while Q ≠ ∅ do
10             Remove an arc (X_i, X_j) out of Q
               ▷ Make (X_i, X_j) arc or bounds consistent:
11             if REVISE(X_i, X_j) modified D_i then
                   ▷ Enqueue the arcs toward X_i:
12                 Q ← Q ∪ {(X_k, X_i) | C_ki ∈ 𝒞, k ≠ j}
13             end if
14         end while
15         if not exists empty D_i ∈ 𝒟 then
               ▷ Proceed to the next level:
16             if ℓ = n then
17                 return success
18             else if DFS_CONS(ℓ + 1) = success then
19                 return success
20             end if
21         end if
           ▷ Restore the previous state of domains:
22         {D_1, ..., D_n} ← {D'_1, ..., D'_n}
23     end for
24     return failure
25 end function
```

Figure 3: A search method that maintains consistency

```
function CONS
    Q ← {(X_i, X_j) | C_ij ∈ 𝒞}
    while Q ≠ ∅ do
        Remove an arc (X_i, X_j) out of Q
        if REVISE(X_i, X_j) modified D_i then
            Q ← Q ∪ {(X_k, X_i) | C_ki ∈ 𝒞, k ≠ j}
        end if
    end while
end function
```

Figure 4: A pure coarse-grained propagation algorithm

In order to get the aggregate T_{path} time, we are going to compute the overall propagation and store-restore time for a search tree *path*, which is a route from the root of the tree ($\ell = 1$) to any of its leaves ($\ell = n$). This means that we will study the overall time of DFS_CONS(1), DFS_CONS(2), ..., DFS_CONS(n) consecutive calls, each of them executing only one iteration of the **for** loop in line 6. The overall path time is at most

$$\begin{aligned} T_{\text{path}} &= \sum_{\ell=1}^{n} T_{\text{path}}(\ell) \\ &= \sum_{\ell=1}^{n} T_{\text{prop}}(\ell) + \sum_{\ell=1}^{n} T_{\text{store}}(\ell) + \sum_{\ell=1}^{n} T_{\text{restore}}(\ell) + \sum_{\ell=1}^{n} T_{\text{const}} \\ &= \sum_{\ell=1}^{n} T_{\text{prop}}(\ell) + 2 \cdot \sum_{\ell=1}^{n} T_{\text{store}}(\ell) + n \cdot T_{\text{const}}, \end{aligned} \tag{1}$$

as T_{const} remains the same for each ℓ, and, as previously explained, $T_{\text{store}} = T_{\text{restore}}$.

This formula applies both to maintaining arc and bounds consistency algorithms. Nevertheless, according to the following table, there are some differentiations that are going to be elaborated on in the following sections.

Path time terms	$\sum_{\ell=1}^{n} T_{\text{prop}}(\ell)$	$2\sum_{\ell=1}^{n} T_{\text{store}}(\ell)$	$n \cdot T_{\text{const}}$
Maintaining AC	n^2d^3	$2nd$	$n \cdot$ constant
Maintaining BC	—"—	$2n^2$	—"—
	Section 3.2	Sections 3.3.1 and 3.3.2	

3.2 The Constraint Propagation Aggregate Complexity

Consistency enforcement algorithms are divided into two large categories: the *coarse-grained* and *fine-grained* algorithms.[5] The best algorithms from the two categories have been proven to have equal time complexities.[7] Therefore, without loss of generality, in order to study consistency enforcement as a whole, it suffices to simply focus on a typical coarse-grained algorithm, such as CONS in Figure 4.

CONS is initially called by DFS_CONS (Figure 3, lines 2–4) before actual search begins. The other propagation section (Figure 3, lines 8–14) is also a CONS extension: These lines keep executing the CONS **while** loop by inserting some more arcs into the Q.

By replacing CONS call (line 3) by its pseudocode in Figure 4, we are able to compute the overall time for the two propagation sections (lines 2–4 and 8–14) of DFS_CONS as the product of the number (E_{total}) of the inserted-removed arcs out of the Q and the time that REVISE takes.

We may have at most $E_{\text{total}} = n^2 \cdot d$ entry operations into the queue Q, where n^2 denotes the maximum number of the arcs (X_i, X_j) with $X_i, X_j \in \mathscr{X}$. After all, each specific arc (X_i, X_j) is initially inserted into the queue and also when a value is deleted out of D_j. Therefore, a specific arc is inserted at most

$1 + d \approx d$ times into the queue, as a value cannot be deleted more than once while descending a search tree path.

In a search tree path, the domains gradually shrink, until they contain just one value in the last level or until a domain is "wiped out." An arc (X_i, X_j) is enqueued when REVISE deletes a value from D_j, and also when X_j is assigned a value. An assignment is equivalent to deleting all the values in D_j, apart from one.

To conclude, we may have at most d deletions of values out of a domain, which can enqueue a specific arc. In sum, we may invoke at most d REVISE calls for a specific arc.

Following Section 1.3, a REVISE call takes approximately d^2 elementary steps. Overall, the propagation part of DFS_CONS will take approximately

$$\begin{aligned}\sum_{\ell=1}^{n} T_{\text{prop}}(\ell) &= E_{\text{total}} \cdot d^2 \\ &= n^2 d \cdot d^2 \\ &= n^2 d^3, \end{aligned} \tag{2}$$

which is the product of how many insertions we may have into the queue (E_{total}) and the REVISE function operations needed when an arc is popped out of the queue (d^2).

The same reasoning applies to faster—yet more complex—propagation algorithms.[7] The only difference is that these algorithms implement faster REVISE functions that still take the same time either for AC or BC.

Again, the important thing for the current theoretical analysis is that, in the worst case, the propagation time complexity remains the same, either while enforcing AC or BC. However, there are significant differences regarding the domains store and restore mechanism.

3.3 Backup and Restore Aggregate Complexity

In the general case, constraint propagation cannot guide us directly to a solution. However, it can be a critical component of a backtracking search method: each assignment made is followed by consistency enforcement and each consistency enforcement is followed by an assignment.

If the constraints are violated, the last assignment is undone. This is a constant-time operation in a consistency-enforcement-free search method. But while a search method maintains consistency, the undo operation involves not only undoing an assignment, but also restoring the domains affected by the consistency enforcement after the assignment.

3.3.1 Storing Domains while Maintaining Arc Consistency

AC enforcement may remove every value out of the domains of the n variables. The maximum domain size is d; hence, we may have at most nd value removals. As we descend a search tree path (from $\ell = 1$ to n), each value can be only removed and not added back to a domain. Thus, the total values removed and

stored for backtracking purposes in a single path is also bounded by

$$\sum_{\ell=1}^{n} T_{\text{store}}^{\text{AC}}(\ell) = nd\,, \tag{3}$$

which is the number of all the domain values in a CSP.

Example 3. Let us have four constrained variables X_1, X_2, X_3, X_4 with domains $D_1 = \{3, 4\}$, $D_2 = \{3, 5, 6\}$, $D_3 = \{0, 1, 2, 3, 4, 5\}$, and $D_4 = \{0, 2, 4, 6, 8, 10\}$. The constraints are $X_1 \neq X_2$, $X_1 \neq X_3$, $X_2 \neq X_3$, and $X_4 = 2X_3$.

The following table contains the changes that take place in the above domains, while searching for a solution to the problem.

Assignments	**Updates in domains**		
	D_2	D_3	D_4
$D_1 \leftarrow \{3\}$	~~3~~, 5, 6	0, 1, 2, ~~3~~, 4, 5	0, 2, 4, ~~6~~, 8, 10
$D_2 \leftarrow \{5\}$		0, 1, 2, ~~3~~, 4, ~~5~~	0, 2, 4, ~~6~~, 8, ~~10~~
$D_3 \leftarrow \{0\}$			0, ~~2~~, ~~4~~, ~~6~~, ~~8~~, ~~10~~
$D_4 \leftarrow \{0\}$			

Searching for a solution includes an assignment (first column) and enforcing consistency to the rest of the domains.

First, in the first row, we make the assignment $D_1 \leftarrow \{3\}$. As $X_1 \neq X_2$, we should remove 3 out of D_2. Similarly, in the same row, we remove 3 out of D_3 as the second constraint is $X_1 \neq X_3$. And as $X_4 = 2X_3$ and $2 \cdot 3 = 6$, we also remove 6 out of D_4.

This was a practical example of arc consistency enforcement after an assignment takes place. We are still at the first level of the search tree.

As we proceed to the second row of the table, we make the assignment $D_2 \leftarrow \{5\}$. When we make an assignment, we proceed one level deeper into the search tree. Every assignment is followed by constraint propagation. In our case, we enforce arc consistency. As $X_2 \neq X_3$, we should remove 5 out of D_3. And as $2 \cdot 5 = 10$, we remove 10 out of D_4.

In the third row, we make the assignment $D_3 \leftarrow \{0\}$. The values 2, 4, and 8 are removed out of D_4, as they do not have any support in D_3 anymore.

The last row is trivial, as we assign $\{0\}$, containing the only remaining value, to D_4.

This was an example on how assignments interchange with constraint propagation during search. In the case of arc consistency constraint propagation, the domains eventually lose all their values. This is done gradually, while traversing the search tree levels. As we should be able to restore the domains in the state that they were in each search tree level, while descending a search tree path, we need to store every value of every domain (nd values).

3.3.2 Storing Domains while Maintaining Bounds Consistency

Bounds consistency can alter only the *bounds* of a domain. In order to store the previous bounds of a domain, we need 2 operations: to record the domain's lower bound and to record the domain's upper bound. At a search path node of level ℓ, the 2 operations can be repeated for every variable's domain; except

for the variables that have been already instantiated, i.e. the variables having only one value in their domains.

These domains are excluded because there are not any other values in them that can be removed; if the last value is removed, we do not proceed, and we backtrack to a previous search tree level. In a search level ℓ, the instantiated variables are at least $\ell - 1$. Therefore, the uninstantiated variables are at most $n - \ell + 1$. The overall time needed to store the initial domains in a search tree node in level ℓ is

$$T^{\mathrm{BC}}_{\mathrm{store}}(\ell) = 2(n - \ell + 1)\,, \tag{4}$$

which is the product of the two operations needed to store the two bounds of a variable, and the number of uninstantiated variables.

For all the nodes of the search tree path it holds

$$\begin{aligned}\sum_{\ell=1}^{n} T^{\mathrm{BC}}_{\mathrm{store}}(\ell) &= \frac{n}{2}\left(T^{\mathrm{BC}}_{\mathrm{store}}(1) + T^{\mathrm{BC}}_{\mathrm{store}}(n)\right)\\ &= \frac{n}{2}(2 \cdot n + 2 \cdot 1)\\ &= n(n+1) \approx n^2\,. \end{aligned} \tag{5}$$

After all, $T^{\mathrm{BC}}_{\mathrm{store}}(\ell)$ is an arithmetic progression a_ℓ. The sum of its n first terms is known to be $\frac{n}{2}(a_1 + a_n)$.

Example 4. Let us consider the same constraint satisfaction problem as in the previous Example 3.

The following table depicts the state of the domains during search. Each row corresponds to a search tree level. The table is different from the one in Example 3, in the sense that it does not contain every value of every domain, but only their *bounds*.

Assignments	**Updates in domains bounds**					
	D_2		D_3		D_4	
	min	max	min	max	min	max
$D_1 \leftarrow \{3\}$	5	6				
$D_2 \leftarrow \{5\}$			0	4	0	8
$D_3 \leftarrow \{0\}$					0	0
$D_4 \leftarrow \{0\}$						

Again, the assignments interchange with constraint propagation. After the first assignment $D_1 \leftarrow \{3\}$, we have to enforce *bounds* consistency. This means that the minimum and maximum values of every domain should have supports to the other constrained variables. If a bound of a domain does not have any support, it is trimmed.

The initial minimum value of D_2 is 3. But as $X_1 \neq X_2$ and $D_1 = \{3\}$, this value is not supported. Therefore, it should be removed out of D_2 and 5 becomes its new minimum value.

Then, we make the assignment $D_2 \leftarrow \{5\}$. As it holds that $X_2 \neq X_3$, the upper bound of D_3 which is 5, is not supported anymore. That is why in the second row of the table, $\max D_3$ has been trimmed to 4. Subsequently, due to the $X_4 = 2X_3$ constraint and as the maximum value 10 of D_4 is not supported now, we delete it, and 8 becomes the new $\max D_4$.

In the third row, we assign $\{0\}$ to D_3. In this case, $\max D_4$ should become 0 too, as this is the only supported value through the $X_4 = 2X_3$ constraint.

This example illustrates that, in every search tree level, we need to store only the bounds of the domains of the unassigned constrained variables, which is the meaning of the above equation (4).

3.4 Will AC or BC be faster?

The answer to this question is unknown before we actually start and finish solving a given arbitrary CSP. There is not any exact mathematical form to know a priori how much time each search methodology will take either while maintaining AC or BC.

Nevertheless, we can bound the time needed by these search methodologies using the above equations to compute the respective *path* times $T^{\text{AC}}_{\text{path}}$ and $T^{\text{BC}}_{\text{path}}$. These two path times allow us not to compute the exact times for AC and BC (that will be simply denoted as TIME_{AC} and TIME_{BC} in the rest of the paper) but at least to get the respective upper bounds $\text{TIME}_{\text{AC BOUND}}$ and $\text{TIME}_{\text{BC BOUND}}$.

Proposition 1. If $n < d$, then $\text{TIME}_{\text{AC BOUND}} > \text{TIME}_{\text{BC BOUND}}$, else if $n > d$, then $\text{TIME}_{\text{AC BOUND}} < \text{TIME}_{\text{BC BOUND}}$.

Proof. TIME_{AC} and TIME_{BC} is bounded by T_{path} if we multiply it by the maximum number of paths. The maximum number of paths is equal to the maximum number of leaves d^n. Therefore,

$$\text{TIME}_{\text{AC BOUND}} = d^n \cdot T^{\text{AC}}_{\text{path}}\,, \tag{6}$$

$$\text{TIME}_{\text{BC BOUND}} = d^n \cdot T^{\text{BC}}_{\text{path}}\,. \tag{7}$$

By combining (1) and (2) we get

$$T_{\text{path}} = n^2 d^3 + 2\sum_{\ell=1}^{n} T_{\text{store}}(\ell) + n \cdot T_{\text{const}}. \tag{8}$$

We specialize the above equation for AC and BC via (3) and (5).

$$T^{\text{AC}}_{\text{path}} = n^2 d^3 + 2nd + n \cdot T_{\text{const}}, \tag{9}$$

$$T^{\text{BC}}_{\text{path}} = n^2 d^3 + 2n^2 + n \cdot T_{\text{const}}, \tag{10}$$

which leads to Proposition 1, because

$$\begin{aligned} & n < d \\ \Leftrightarrow\ & 2n \cdot n < 2n \cdot d \\ \Leftrightarrow\ & n^2 d^3 + 2n^2 + n\,T_{\text{const}} < n^2 d^3 + 2nd + n\,T_{\text{const}} \\ \Leftrightarrow\ & T^{\text{BC}}_{\text{path}} < T^{\text{AC}}_{\text{path}} \\ \Leftrightarrow\ & d^n T^{\text{BC}}_{\text{path}} < d^n T^{\text{AC}}_{\text{path}} \\ \Leftrightarrow\ & \text{TIME}_{\text{BC BOUND}} < \text{TIME}_{\text{AC BOUND}}. \end{aligned}$$

□

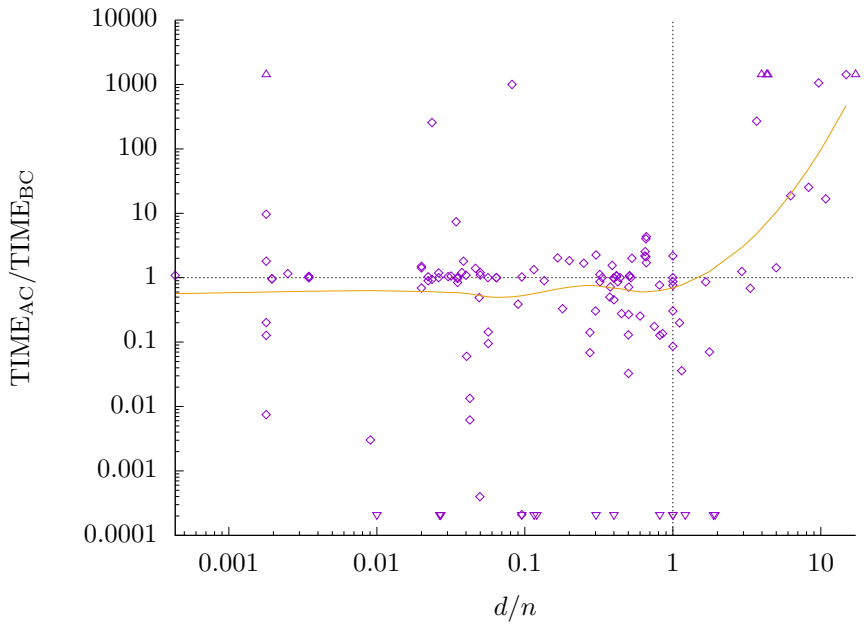

Figure 5: The time needed to solve the CSPs while maintaining AC divided to the time spent while maintaining BC

4 Empirical Evaluations

All the above theory was inspired by observations while solving artificial and real-life constraint satisfaction problems. To test the theoretical results of this work in practice, we consider all standard CSP instances taken from the First XCSP3 Constraint Mini-Solver Competition.[11] The specific instances used in the *mini-solver* track are available under the respective link in the competition site `http://www.cril.univ-artois.fr/XCSP17`

Table 1 displays raw experimental results, while Figure 5 depicts them graphically. But, before going through all these empirical results, let us describe how one can reproduce them.

4.1 Methodology

In order to make comparisons, we had to employ two different solvers: one that maintains arc consistency (AC) and another that maintains bounds consistency (BC). Therefore, we took the open source Naxos Solver[21] and created its AC and BC variants.

Note that the original Naxos Solver implements several consistency levels for various constraints. Consequently, we created two sets of patches, one that implements pure arc consistency and another for pure bounds consistency for every constraint employed. All patches are available at `https://github.com/pothitos/ACvsBC-Solver-Patches`

Similarly to the theory of this work, we considered only binary constraints (that apply between two constrained variables) to simplify consistency enforce-

Table 1: CSP attributes and solution times while maintaining AC and BC

CSP	n	d	TIME$_{\text{AC}}$	TIME$_{\text{BC}}$	CSP	n	d	TIME$_{\text{AC}}$	TIME$_{\text{BC}}$
aim-50-2-0-unsat-2	**50**	2	0.71	**0.65**	MarketSplit-09	**152**	100	570.38	**270.14**
AllInterval-007	**25**	13	0.07	0.07	MarketSplit-10	**151**	100	243.32	**142.11**
AllInterval-012	**45**	23	0.14	**0.13**	MultiKnapsack-1-03	235	**2536**	14.62	**0.87**
AllInterval-016	**61**	31	0.20	**0.19**	MultiKnapsack-1-5_X2	239	**4106**	X	**95.23**
AllInterval-035	**137**	69	**0.65**	0.91	MultiKnapsack-2-16	274	**1181**	X	**76.94**
AllInterval-050	**197**	99	**1.42**	5.27	MultiKnapsack-2-21	342	**1361**	X	**46.15**
AllInterval-080	**317**	159	**6.65**	203.52	MultiKnapsack-2-22	342	**1501**	X	**163.88**
bdd-15-21-2-2713-79-08	**21**	2	41.39	**40.37**	MultiKnapsack-2-41	136	**1126**	73.21	**2.90**
bdd-15-21-2-2713-79-16	**21**	2	**2326.85**	X	MultiKnapsack-2-48	180	**1126**	811.88	**43.36**
bqwh-15-106-35_X2	**106**	6	**0.43**	4.53	Nonogram-018-table	**576**	2	3.14	**2.98**
bqwh-15-106-36_X2	**106**	6	**0.21**	1.46	Nonogram-035-table	**576**	2	**2.48**	2.49
bqwh-18-141-09_X2	**141**	6	**7.99**	597.87	Nonogram-096-table	**576**	2	5.79	**5.73**
bqwh-18-141-31_X2	**141**	7	**0.33**	828.42	Nonogram-168-table	**400**	1	1.20	**1.04**
bqwh-18-141-83_X2	**141**	6	**5.18**	837.63	Nonogram-177-table	**1024**	2	**2.57**	2.69
color_X2	**500**	5	**68.21**	X	Nonogram-180-table	**1024**	2	**32.95**	34.15
ColouredQueens-03	**9**	3	0.01	0.01	Pb-queen-0974553	**1137**	39	25.70	**3.47**
composed-25-01-25-3	**33**	10	0.09	**0.04**	pigeonsPlus-07-05	**42**	7	17.76	**8.76**
composed-25-01-25-4	**33**	10	**0.09**	X	pigeonsPlus-08-04	**40**	8	52.36	**28.48**
composed-25-10-20-5	**105**	10	**0.31**	1481.76	pigeonsPlus-09-03	**36**	9	239.93	**143.16**
composed-75-01-25-6	**83**	10	**0.22**	X	Primes-10-20-3-3	213	**784**	10.79	**0.04**
cril-5_X2	42	**81**	**55.06**	X	Primes-10-60-3-3	444	**784**	**46.51**	659.36
Crossword-m1c-lex-h1501	**225**	26	**11.34**	X	Primes-15-20-2-5	219	**2116**	168.98	**0.16**
Crossword-m1c-ogd-h2310	**529**	26	**40.89**	83.52	Primes-20-40-2-1	241	**3574**	71.23	**0.05**
Crossword-m1c-uk-vg-4-8	**32**	26	**11.39**	14.81	PropStress-0020	**293**	24	1297.90	**1.30**
Crossword-m1c-words-p20	**81**	26	0.65	**0.58**	qwh-10-57-7_X2	**100**	5	0.11	**0.10**
driverlogw-01c	**71**	4	0.02	0.02	rand-2-23-23-253-131-0	23	23	**673.35**	X
driverlogw-02c	**301**	8	**110.01**	X	rand-2-23-23-253-131-1	23	23	**663.37**	X
driverlogw-04c	**272**	11	**3.03**	50.28	rand-2-30-15-306-230f-09	**30**	15	**8.51**	65.62
driverlogw-08c	**408**	11	**263.16**	X	rand-2-40-11-414-020-23	**40**	11	**46.38**	328.02
driverlogw-08cc	**408**	11	**254.62**	X	rand-2-40-11-414-020-35	**40**	11	**4.67**	68.40
Dubois-021	**63**	2	149.08	**140.53**	rand-5-12-12-200-12442-38	12	12	**642.24**	846.58
Dubois-022	**66**	2	303.95	**291.51**	rand-5-12-12-200-t95-3	12	12	**701.10**	803.31
ehi-85-297-30	**297**	7	84.50	**0.33**	rand-5-2X-05c-15	12	12	**558.94**	1837.32
ehi-85-297-98	**297**	7	**0.32**	0.34	Renault	**101**	42	3.87	**3.66**
ehi-90-315-13	**315**	7	**0.27**	0.30	Renault-medium-pos	**148**	20	**0.27**	0.30
ehi-90-315-37	**315**	7	0.34	**0.33**	Renault-megane-pos	**99**	42	**3.05**	3.53
geometric-50-20-d4-75-03	**50**	20	**0.50**	0.51	Renault-mgd	**101**	42	3.74	**3.48**
geometric-50-20-d4-75-46	**50**	20	**13.79**	X	Renault-small	**139**	16	0.08	**0.06**
geometric-50-20-d4-75-54	**50**	20	**0.30**	0.66	Renault-souffleuse	**32**	12	**0.01**	0.02
jnh-012	**100**	2	0.17	**0.12**	RenaultMod-09	**111**	42	**740.11**	1032.59
jnh-213	**100**	2	0.12	**0.08**	Sat-flat200-06-dual	**2237**	4	470.44	**261.04**
jnh-302	**100**	2	**0.09**	0.13	Sat-flat200-14-dual	**2237**	4	**1.34**	179.40
Kakuro-easy-015-sumdiff	**194**	9	0.07	**0.05**	Sat-flat200-32-dual	**2237**	4	130.75	**13.55**
Kakuro-easy-079-sumdiff	**344**	9	0.13	**0.11**	Sat-flat200-55-dual	**2237**	4	**146.12**	1147.39
Kakuro-easy-084-ext	**240**	9	0.48	**0.40**	Sat-flat200-65-sum	**6911**	3	127.33	**117.12**
Kakuro-easy-109-ext	**256**	9	**0.97**	1.15	Sat-flat200-67-dual	**2237**	4	**138.31**	688.10
Kakuro-easy-150-ext	**256**	9	0.79	**0.78**	Sat-flat200-80-dual	**2237**	4	X	**1574.29**
Kakuro-easy-164-sumdiff	**344**	9	0.30	0.30	SchurrLemma-mod-012-9	**12**	9	**6.88**	39.40
Kakuro-hard-179-sumdiff	**996**	9	**2.02**	666.57	SchurrLemma-mod-015-9	**15**	9	**9.45**	37.19
Kakuro-medium-016-ext	**140**	9	0.15	0.15	SchurrLemma-mod-020-9	**20**	9	**11.85**	42.86
Kakuro-medium-020-ext	**140**	9	0.09	0.09	SchurrLemma-mod-030-9	**30**	9	**17.74**	58.30
Kakuro-medium-055-sumdiff	**234**	9	0.09	**0.05**	SchurrLemma-mod-050-9	**50**	9	**33.20**	100.52
Kakuro-medium-162-ext	**256**	9	**13.81**	14.25	SchurrLemma-mod-100-9	**100**	9	**116.89**	301.21
Langford-3-05	**25**	11	0.12	0.12	Subisomorphism-A-15	180	**200**	**2.61**	13.11
Langford-4-04	**28**	9	**0.13**	0.15	Subisomorphism-g07-g39	**20**	1	0.06	**0.05**
Langford-4-05	**35**	14	0.17	0.17	Subisomorphism-g08-g31	30	**100**	**4.83**	7.06
MagicHexagon-02-0000	**18**	7	0.14	**0.09**	Subisomorphism-g10-g35	41	**120**	0.05	**0.04**
MagicSquare-3-sum	**17**	9	0.02	**0.01**	Subisomorphism-si2-b09-m200-02	40	**200**	0.30	**0.21**
MagicSquare-3-table	9	9	0.01	0.01	Subisomorphism-si6-b03-m800-07	480	**800**	**1.23**	1.43
MagicSquare-4-table	16	16	0.24	**0.11**	TravellingSalesman-20-076_X2	61	**70**	**50.54**	1407.10
MagicSquare-5-table	25	25	**139.94**	1627.44	TravellingSalesman-20-142_X2	61	**115**	**1640.18**	X
MarketSplit-03	**151**	100	1149.24	**264.75**	TravellingSalesman-25-003_X2	**76**	62	**190.14**	X
MarketSplit-05	**153**	99	674.29	**309.90**	TravellingSalesman-25-066_X2	**76**	62	**28.76**	224.92
MarketSplit-07	**152**	100	595.51	**148.30**	TravellingSalesman-4-20-001-a4_X2	**61**	52	**60.84**	448.12
MarketSplit-08	**154**	100	292.73	**115.03**	TravellingSalesman-4-20-727-a4_X2	61	**74**	**708.48**	X

ment. Therefore, we binarized the global constraints (that apply to more than two variables) that exist in some CSP instances by substituting them by groups of equivalent binary constraints.

Finally, it is worth noting that, in order to be more accurate, the illustrated CSP parameters n and d (number of constrained variables and maximum domain cardinality in the CSP) are not taken directly from the CSP definition; they are reported by the solver itself. Consequently, n is reported only after the binarization of the constraints has been completed, possibly by adding more constrained variables.

Also, we enforce bounds consistency for the first time, before displaying the maximum domain cardinality d. This means for example that if we have two constrained variables X_1 and X_2, with $X_1 \leq X_2$ and $D_1 = \{1, 2, \ldots, 100\}$ and $D_2 = \{25, 26, \ldots, 50\}$, the maximum cardinality will not be computed as 100. Bounds consistency will be enforced first, and D_1 will be limited to $\{1, 2, \ldots, 50\}$. The maximum domain cardinality d will be eventually displayed as 50. In this way we "normalize" redundant domains.

4.2 Execution

In order to construct Table 1 with the experimental results, we follow the above methodology and display n and d for each CSP instance. If n is greater than d, we display it bold, else d is displayed bold. In theory, when d is greater than n, we expect that maintaining bounds consistency is more efficient than maintaining arc consistency.

Using the above methodology, we created two separate solvers, one that maintains arc consistency and one that maintains bounds consistency. Each of them was assigned to solve the First XCSP3 Constraint Mini-Solver Competition CSPs.[11] Each CSP instance has to be solved within 40 minutes according to the competition requirements. If a solver cannot solve an instance within this time frame, it is marked with an "X" in the table. Otherwise, the elapsed time in seconds is written. Please note that only the CSP instances that were solved at least from one solver are displayed in the table.

4.3 Visualization

In order to make comparisons more easily, we depicted graphically the ratio $\text{TIME}_{\text{AC}}/\text{TIME}_{\text{BC}}$ versus d/n in Figure 5 using the $\diamond$ symbol.

When the AC solver does not produce a solution, we have an undefined TIME_{AC} denoted as "X" in the table. In the figure, the corresponding point is depicted with a $\triangle$ symbol. This represents a very high $\text{TIME}_{\text{AC}}/\text{TIME}_{\text{BC}}$ ratio, which means that maintaining BC is much more efficient than AC in this case.

On the other hand, when TIME_{BC} is "X," the ratio $\text{TIME}_{\text{AC}}/\text{TIME}_{\text{BC}}$ is depicted with a $\triangledown$ symbol. This denotes a very low ratio, which means that AC is much more efficient than BC in this case.

It may be obvious that the above $\triangle$ and $\triangledown$ points do not correspond to real values. They are used in the margins of Figure 5 to represent marginal ratios, as described above.

As the $\diamond$ points in the figure are somehow sparse, the results become more intuitive if we draw a smooth curve between them. Therefore, the curve in Figure 5 has been derived by the LOESS method[9,31] and is representative of

the $\diamond$ points. LOESS method does not consider the marginal $\triangle$ and $\triangledown$ points because they do not depict real values.

4.4 Observations

In Figure 5 we compare the times for solving a CSP instance via maintaining AC and BC. A first conclusion is that BC can be better than AC for many instances. This is an important observation, as, due to the fact that AC enforces a stronger consistency level than BC, and both AC and BC have equal worst-case complexities (Lemma 1), there is the misconception that AC is always better than BC.

However, the conclusion about the occasional superiority of BC over AC has no practical use, if we do not know when it happens. We have to find the appropriate conditions to know a priori if a CSP instance will be solved faster by maintaining AC or BC.

In theory (Proposition 1) the relation between n and d defines the relation between the upper limits of TIME_{AC} and TIME_{BC}. To put it simply, the d/n ratio affects the $\text{TIME}_{\text{AC}}/\text{TIME}_{\text{BC}}$ ratio, and this is evident in practice in Figure 5: On average, $\text{TIME}_{\text{AC}}/\text{TIME}_{\text{BC}} < 1$ if $d/n < 1$ and $\text{TIME}_{\text{AC}}/\text{TIME}_{\text{BC}} > 1$ if $d/n > 1$. This becomes clearer if we observe the smooth curve constructed by the LOESS method, which represents the "average" of the $\diamond$ points.[9]

Of course, there is some deviation between our theoretic expectations and the observed results. This is due to the fact that in theory we studied the worst case of complete search trees for both maintaining AC and BC, while in practice the two methodologies may produce incomplete search trees that are different between them.

Regarding the $\triangle$ points (that represent the cases when only the maintaining BC method found a solution while maintaining AC did not find one) they are apparently more on the right side, i.e. when $d/n > 1$. On the other hand, the $\triangledown$ points are gathered mostly on the left side of Figure 5. This means that for $d/n < 1$, the maintaining BC methodology is usually not only less efficient than AC, but it may produce no solution for a CSP, while AC is able to solve it.

5 Conclusions and Future Perspectives

The main contribution of this work is to give focus on the weaker consistency levels (BC) in Constraint Programming and to highlight their advantages over "stronger" consistency levels (AC). If we take it for granted that arc and bounds consistency have both equal asymptotic time complexities, then two questions arise.

1. Why BC is often used in practice in Constraint Programming solvers?
2. When should we prefer BC over AC?

In current bibliography, answers to the first question are scarce and only based on unpublished empirical observations. In any case, one can answer to the first question by conducting experiments and finding examples where BC is more efficient than AC. Indeed, in this work, we experimented with a broad

range of official CSPs and found many cases where BC is more efficient in practice.

The second question is more difficult, as it is addressed for every possible ad hoc CSP. Our approach to answer it included the following steps.

- Introduce the algorithms for arc and bounds consistency enforcement and prove that they take the same time in the worst case.
- Introduce a basic backtracking search algorithm and the search tree and search path notions.
- Integrate consistency enforcement algorithms into the backtracking search method.
- Compute the overall time complexity while descending a search tree path and find the differentiations between maintaining AC and BC.
- Project the complexity to traverse a search path to the overall search tree complexity.

Following this approach, we produced some tight upper limits for AC and BC time complexities in the context of search methods. We defined a criterion which, based on the attributes of a CSP, predicts which of the two methodologies is likely to solve it faster.

This new criterion gives us the freedom to select the consistency level (AC or BC) just before solving a specific CSP. We are not obliged to use default consistency levels when we build a Constraint Programming solver anymore. We are now able to tailor the AC vs. BC selection to the particular parameters of each CSP and thus make the overall search process more efficient.

In the future, this work can be naturally extended to answer the question why even higher consistency levels than AC are "seldom used in practice".[3] This is another paradox, as there are a lot of very important publications for sophisticated higher consistency levels. Just like in this work, we should develop criteria about *when* to use higher consistency levels than AC and not completely ignore them.

Another natural future extension of this work will be to compare the maintenance of *generalized* arc and bounds consistencies during search, which are enforced to non-binary constraint networks. In this work, we considered only binary constraints, i.e. only constraints between two variables. This was done for the sake of simplicity, as every constraint involving more than two variables can be converted to binary constraints.[25] After all, the notion of the *arc*, e.g. (X_1, X_2), includes only two variables.

On the other hand, n-ary constraints with $n > 2$, i.e. constraints that involve more than two variables, are quite common in practice and can be exploited to speed up search. Such constraints are often expressive in the sense that it is more elegant for example to mention AllDifferent(X_1, X_2, X_3) than $X_1 \neq X_2 \wedge X_2 \neq X_3 \wedge X_3 \neq X_1$.

For n-ary constraints, we enforce either generalized arc consistency (GAC) or generalized bounds consistency. It would be interesting to see if the behavior of maintaining AC vs. BC during search remains the same for their generalized variants.

In 1997, Eugene Freuder, a Constraint Programming pioneer, stated that its "holy grail" is that the user simply states the problem and the computer solves it.[12] This, obviously, emphasizes on user experience. Today, after two decades of theoretical advances, the community still pursuits this "holy grail".[13] If we want to contribute toward this direction in the future, we should integrate and test the existing theory (e.g. about various consistency levels, search methodologies, etc.) into user-friendly solvers and take the decision to use a common testbed with emphasis on real-life CSPs over artificial ones with obfuscated modelings.

References

1. Ayub MA, Kalpoma KA, Proma HT, Kabir SM, Chowdhury RIH. Exhaustive study of essential constraint satisfaction problem techniques based on N-queens problem. In *ICCIT*:1–6 IEEE 2017.

2. Balafrej A, Bessiere C, Paparrizou A. Multi-armed bandits for adaptive constraint propagation. In *IJCAI 2015*:290–296 2015.

3. Balafrej A, Bessiere C, Paparrizou A, Trombettoni G. Adapting consistency in constraint solving. In *Data Mining and Constraint Programming* (Bessiere C, De Raedt L, Kotthoff L, Nijssen S, O'Sullivan B, Pedreschi D, eds.):226–253 Springer 2016.

4. Barták R, Salido MA, Rossi F. Constraint satisfaction techniques in planning and scheduling. *Journal of Intelligent Manufacturing.* 2010;21(1):5–15.

5. Bessiere C. Constraint propagation. In Rossi *et al.*[26]:29–83.

6. Bessiere C. Constraint reasoning. In Marquis *et al.*[18]:153–183.

7. Bessiere C, Régin JC, Yap RHC, Zhang Y. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence.* 2005;165(2):165–185.

8. Bulatov AA. A dichotomy theorem for nonuniform CSPs. In *FOCS 2017*:319–330 IEEE 2017.

9. Cleveland WS, Grosse E, Shyu WM. Local regression models. In *Statistical Models in S* (Chambers JM, Hastie TJ, eds.) ch. 8:309–376 Chapman & Hall 1991.

10. Cohen DA, Jeavons PG. The power of propagation: When GAC is enough. *Constraints.* 2017;22(1):3–23.

11. First XCSP3 competition. `http://xcsp.org/call2017.pdf` 2017.

12. Freuder EC. In pursuit of the holy grail. *Constraints.* 1997;2(1):57–61.

13. Freuder EC. Progress towards the holy grail. *Constraints.* 2018;23(2):158–171.

14. Howell I, Woodward RJ, Choueiry BY, Bessiere C. Solving Sudoku with consistency: A visual and interactive approach. In *IJCAI 2018*:5829–5831 2018.

15. Lallouet A, Moinard Y, Nicolas P, Stéphan I. Logic programming. In Marquis *et al.*[18]:83–113.

16. Li H, Li R, Yin M. Saving constraint checks in maintaining coarse-grained generalized arc consistency. *Neural Computing and Applications.* 2019;31(1):499–508.

17. Mackworth AK. Constraint-based design of embedded intelligent systems. *Constraints.* 1997;2(1):83–86.

18. Marquis P, Papini O, Prade H, eds. *A Guided Tour of Artificial Intelligence Research, Volume II: AI Algorithms.* Springer 2020.

19. Petke J. *Bridging Constraint Satisfaction and Boolean Satisfiability.* Springer 2015.

20. Pisinger D, Ropke S. Large neighborhood search. In *Handbook of Metaheuristics* (Gendreau M, Potvin JY, eds.):99–127 Springer 2019.

21. Pothitos N. Naxos Solver. `http://github.com/pothitos/naxos` 2018.

22. Pothitos N, Stamatopoulos P. Building search methods with self-confidence in a constraint programming library. *IJAIT.* 2018;27(4):1860003.

23. Pothitos N, Stamatopoulos P, Zervoudakis K. Course scheduling in an adjustable constraint propagation schema. In *ICTAI 2012*:335–343 IEEE 2012.

24. Puget JF, Van Hentenryck P. A constraint satisfaction approach to a circuit design problem. *Journal of Global Optimization.* 1998;13(1):75–93.

25. Rossi F, Petrie C, Dhar V. On the equivalence of constraint satisfaction problems. In *ECAI 1990*:550–556 1990.

26. Rossi F, Beek P, Walsh T, eds. *Handbook of Constraint Programming.* Elsevier 2006.

27. Smith BM. Modelling. In Rossi *et al.*[26]:377–406.

28. Veksler M, Strichman O. Learning general constraints in CSP. *Artificial Intelligence.* 2016;238:135–153.

29. Wang R, Yap RHC. Arc consistency revisited. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*:599–615 Springer 2019.

30. Wegener I. *Complexity Theory* ch. 2:20. Springer 2005.

31. Wilcox R. The regression smoother LOWESS: A confidence band that allows heteroscedasticity and has some specified simultaneous probability coverage. *Journal of Modern Applied Statistical Methods.* 2017;16(2):29–38.

32. Woodward RJ, Choueiry BY, Bessiere C. A reactive strategy for high-level consistency during search. In *IJCAI 2018*:1390–1397 2018.

33. Zhuk D. A proof of CSP dichotomy conjecture. In *FOCS 2017*:331–342 IEEE 2017.