Empirical Evaluation of the Implicit Hitting Set Approach for Weighted CSPs

Aleksandra Petrova
Department of Computer Science, Universitat Politècnica de Catalunya, spain

Javier Larrosa
Department of Computer Science, Universitat Politècnica de Catalunya, spain

Emma Rollón
Department of Computer Science, Universitat Politècnica de Catalunya, spain

Abstract

SAT technology has proven to be surprisingly effective in a large variety of domains. However, for the *Weighted* CSP problem dedicated algorithms have always been superior. One approach not well-studied so far is the use of SAT in conjunction with the *Implicit Hitting Set* approach. In this work, we explore some alternatives to the existing algorithm of reference. The alternatives, mostly borrowed from related boolean frameworks, consider trade-offs for the two main components of the IHS approach: the computation of low-cost hitting vectors, and their transformation into high-cost cores. For each one, we propose 4 levels of intensity. Since we also test the usefulness of cost function merging, our experiments consider 32 different implementations. Our empirical study shows that for WCSP it is not easy to identify the best alternative. Nevertheless, the cost-function merging encoding and extracting maximal cores seems to be a robust approach.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Weighted CSPs, Implicit Hitting Set

Digital Object Identifier 10.4230/LIPIcs.Soft 2024.2024.

Acknowledgements This work was supported by grant PID2021-122830OB-C43, funded by MCIN/AEI/10.13039/501100011033 and by "ERDF: A way of making Europe".

1 Introduction

The *Weighted CSP* problem (WCSP) is a framework for discrete optimization with many practical applications [7, 24, 25, 4, 7, 24, 25, 4] that has attracted the interest of researchers for decades [8, 19, 1, 3]. In this paper, we focus on the *Implicit Hitting Set Approach* (IHS) for WCSP solving. The idea of the IHS algorithms is to iteratively grow a set of unsatisfiable pieces of the problem (called *cores*) and find if it is possible to solve the problem by avoiding (*i.e*, *hitting*) them. The algorithm terminates when the best way to hit the identified cores incidentally also hits the unidentified cores.

The motivation for our work is that IHS is surprisingly effective for the MaxSAT problem [11, 12, 5], and such success has been lifted to several generalization frameworks such as *Pseudo-boolean* optimization [22, 23], *MaxSMT* [14] and *Answering Set Programming* [21]. Although MaxSAT and WCSP are fairly similar, to the best of our knowledge the only works that consider IHS for WCSP are [13, 18] and, in both cases, a very simple IHS strategy is used.

In this paper, we test the potential of several variations of the original algorithm that have already been proposed and successfully applied in other frameworks. Here, we adapt them to WCSPs and test their effect empirically. We consider four different ways in which to obtain hitting vectors and four different ways in which to transform them into an improved core. Since we also test the effectiveness of *cost-function merging* [18] we end up with $4 \times 4 \times 2 = 32$

Workshop on Discrete Optimization with Soft Constraints.
Editors: John Q. Open and Joan R. Access; Article No. ; pp. :1–:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

different algorithms that we test on several benchmarks.

The experiments show that different algorithms may have a dramatic difference in performance (up to several orders of magnitude). Most importantly, they show that no algorithm systematically dominates all the others, but merging cost-functions following [18] and computing maximal cores seems to be the most robust strategy. This configuration most of the times is the most efficient or is close to the most efficient. Regarding the computation of hitting vectors, greedy heuristics often do not pay off. Although there is no clear winner between optimal and cost-bounded hitting vectors, cost-bounded ones seem to have more potential because it offers more implementation alternatives.

2 Preliminaries

A *Constraint Satisfaction Problem* (CSP) is a pair (X, C) where X is a set of *variables* taking values in a finite domain, and C is a set of *constraints*. Each constraint depends on a subset of variables called *scope*. Constraints are boolean functions that forbid some of the possible assignments of the scope variables. A *solution* is an assignment to every variable that satisfies all the constraints. Solving CSPs is known to be an NP-complete problem [16].

A *Weighted CSP* (WCSP) is a CSP augmented with a set F of *cost functions*. A cost function $f \in F$ is a mapping that associates a cost to each possible assignment of the variables in its scope. The *cost of a solution* is the sum of costs given by the different cost functions. The WCSP problem consists of computing a *solution of minimum cost* w^*.

The IHS approach for WCSP is defined in terms of vectors. The *cost* of vector $\vec{v} = (v_1, v_2, \ldots, v_m)$ is $cost(\vec{v}) = \sum_{i=1}^{m} v_i$. In the (partial) order among same-size vectors, $\vec{u} \leq \vec{v}$, holds iff for each component i we have that $u_i \leq v_i$. If $\vec{u} \leq \vec{v}$ we say that $\vec{v}$ *dominates* $\vec{u}$. We say that a set of vectors $\mathcal{V}$ *dominates* a vector $\vec{u}$, noted $\vec{u} \leq \mathcal{V}$, if there is some $\vec{v} \in \mathcal{V}$ that dominates $\vec{u}$. Further, we say that a set of vectors $\mathcal{V}$ *dominates* a set of vectors $\mathcal{U}$, noted $\mathcal{U} \leq \mathcal{V}$, if $\mathcal{V}$ dominates every vector of $\mathcal{U}$. Given a set of vectors $\mathcal{U}$, a vector $\vec{u} \in \mathcal{U}$ is *maximal* if it is not dominated by any other element of $\mathcal{U}$. The set of maximal vectors in $\mathcal{U}$ is noted $\overline{\mathcal{U}}$. The set of vectors in $\mathcal{U}$ with cost less than w is noted $\mathcal{U}_{(<w)}$. If $\vec{u}$ is *not* dominated by $\mathcal{V}$ we say that it *hits* $\mathcal{V}$. The *minimum cost hitting vector* MHV of $\mathcal{V}$ is a vector that hits $\mathcal{V}$ with minimum cost. It is not difficult to see that MHV reduces to the classic *minimum hitting set problem* [15], which is known to be NP-hard.

In the following, we will consider an arbitrary WCSP (X, C, F) with m cost functions $F = \{f_1, f_2, \ldots, f_m\}$. A *cost vector* $\vec{v} = (v_1, v_2, \ldots, v_m)$ is a vector where each component v_i is associated to cost function f_i, and value v_i must be a cost occurring in f_i. Cost vector $\vec{v}$ *induces* a CSP $(X, C \cup F_{\vec{v}})$ where $F_{\vec{v}}$ denotes the set of constraints $(f_i \leq v_i)$ for $1 \leq i \leq m$ (namely, cost functions are replaced by constraints). If the CSP induced by $\vec{v}$ is satisfiable we will say that $\vec{v}$ is a *solution vector*. Otherwise, we will say that $\vec{v}$ is a *core*. The set of all cores will be denoted *Cores*. An *optimal solution vector* is a solution vector of minimum cost. It is easy to see that the cost of an optimal solution vector is the same as the optimum cost w^* of the WCSP.

3 IHS-based WCSP solving

The IHS approach relies on the following observation that establish a lower bound and an upper bound condition in terms of cores and solutions,

▶ **Observation 1.** *Consider a solution vector* $\vec{h}$ *and a set of cores* $\mathcal{K} \subseteq Cores$*. Then,* $MHV(\mathcal{K}) \leq w^* \leq cost(\vec{h})$.

```
Function IHS-lb(X, C, F)
begin
    K := ∅; lb := 0; ub := ∞ ;
    while lb < ub do
        h⃗ :=MinCostHV(K);
        lb := cost(h⃗);
        if SolveCSP(X, C ∪ F_h⃗) then
            ub := cost(h⃗);
        else
            k⃗ :=ImprCore(X, C, F, ub, h⃗);
            K := K ∪ {k⃗};
        end
    end
    return lb
end
```

```
Function IHS-ub(X, C, F)
begin
    K := ∅; lb := 0; ub := ∞ ;
    while lb < ub do
        h⃗ :=CostBoundedHV(K, ub);
        if h⃗ = NUL then lb := ub;
        else
            if SolveCSP(X, C ∪ F_h⃗) then
                ub := cost(h⃗);
            else
                k⃗ :=ImprCore(X, C, F, ub, h⃗);
                K := K ∪ {k⃗};
            end
        end
    end
    return lb
end
```

Algorithm 1 Two different IHS algorithms for WCSP. Both receive as input a WCSP (X, C, F) and returns the cost of the optimal solution w^*. Function *ImprCore()* receives as input a core $\vec{h}$ and returns a core $\vec{k}$ such that $\vec{h} \leq \vec{k}$. It may also improve the upper bound ub.

All the algorithms discussed in this paper will aim at finding a solution $\vec{h}$ and a (possibly small) set of cores $\mathcal{K}$ such that the two bounds meet (that is, $MHV(\mathcal{K}) = cost(\vec{h})$). This condition corresponds to $\vec{h}$ being optimal and $\mathcal{K}$ being the proof of its optimality. We will refer to this (termination) condition as TC.

Consider the set of all cores with costs less than w^*. We define a *goal core* as a maximal core in that set. That is, the set of goal cores is $\mathcal{GC} = \overline{Cores}_{(<w^*)}$ The following observation rephrases the lower bound part of TC as having a set of cores $\mathcal{K}$ that dominates all maximal goal cores,

▶ **Observation 2.** *A set of cores $\mathcal{K}$ satisfies $MHV(\mathcal{K}) = w^*$ if and only if $\mathcal{GC} \leq \mathcal{K}$, where $\mathcal{GC}$ denotes the set of goal cores.*

Therefore, IHS algorithms must compute a set $\mathcal{K}$ that dominates every goal core (lower bound condition of TC) and an optimal solution (upper bound part of TC).

3.1 Baseline Algorithm

The algorithm proposed in [13] appears at the left of Algorithm 1. It is a loop that maintains three variables: a working set of cores $\mathcal{K}$, a lower bound lb, and an upper bound ub of the optimum. At each iteration, the algorithm computes in $\vec{h}$ the MHV of $\mathcal{K}$ and solves the CSP that it induces. If it is satisfiable the algorithm will stop, else $\vec{h}$ is improved into core $\vec{k}$, which is added to $\mathcal{K}$, and the algorithm goes on. The details for *ImprCore()* will be discussed in Subsection 4.2. For the moment, just note that it returns a core vector $\vec{k}$ such that $\vec{h} \leq \vec{k}$. *ImprCore()* may find solution vectors during its execution. If their cost is smaller than the upper bound, the upper bound will be accordingly updated. Because the emphasis of this algorithm is in the lower bound (the upper bound is only updated if better solutions are found incidentally) we will refer to it as IHS-*lb*.

It is worth noting at this point that each iteration of IHS can be divided into two parts: *i*) the computation of the hitting vector $\vec{h}$ which requires solving an NP-hard optimization problem and *ii*) if $\vec{k}$ is a core, its transformation into a larger core $\vec{k}$ which requires solving a sequence of CSPs, which are NP-complete decision problems. In this paper, we restrict ourselves to the usual choice of computing hitting vectors with a 0/1 IP solver and solving induced CSPs with a SAT solver.

4 Algorithmic Alternatives

4.1 Computation of Hitting Vectors

In the following, we describe some alternatives to alleviate the time spend computing hitting vectors.

4.1.1 Non-optimal Cost-bounded Hitting Vectors

One way to decrease the workload of each iteration is to rely on non-optimal hitting vectors. As suggested in [23], we can replace optimal hitting vectors by hitting vectors of bounded cost. The right side of Algorithm 1 shows this idea. At each iteration, a hitting vector $\vec{h}$ with cost less than ub is obtained. If $\vec{h}$ is a solution the upper bound is updated, else it is improved and added to $\mathcal{K}$. Since the emphasis of this algorithm is in the upper bound, we will refer to it as IHS-ub.

The main advantage of IHS-ub compared to IHS-lb is that iterations are likely to be faster. There are several reasons for this. On the one hand, it is much more efficient to find a bounded hitting vector which is a decision problem, than finding an optimal hitting vector which is an optimization problem. On the other hand, only in the last call of *CostBoundedHV()* the problem will be unsatisfiable which is typically a much more costly task to solve. Another reason is that cost-bounded hitting vectors are likely to have a higher cost (near ub) and therefore obtaining an improved core $\vec{k}$ will not need so many *SolveCSP()* calls (whatever the stopping criteria are). However, the advantage is at the cost of potentially more iterations. On the one hand in IHS-ub, not all iterations end up adding a new core because some iterations only decrease the upper bound. On the other hand, the computed hitting vectors are no longer guaranteed to have cost below w^*, therefore new cores may not contribute towards the lower bound part of TC (that is, $\mathcal{GC} \leq \mathcal{K}$).

4.1.2 Cost-unbounded Hitting Vectors

Although obtaining cost-bounded hitting vectors can be done much more efficiently in practice than obtaining optimal hitting vectors, the problem remains NP-complete and, therefore, may still be time-consuming. As suggested in [11] and subsequently applied in [22], one way to avoid expensive calls is by removing the cost-bound requirement. Obtaining a low-cost hitting vector without requiring the cost to be below a bound can easily be done with an incomplete algorithm.

We consider a greedy algorithm that starts from $\vec{h} = \vec{0}$ and makes a sequence of increments dictated by some greedy criterion until $\vec{h}$ hits every vector in $\mathcal{K}$. Since we want to hit as many cores as possible with the lowest cost, our algorithm selects the increment that minimizes the corresponding ratio. This algorithm is similar to a well-studied greedy algorithm for *vertex covering* [9].

If the resulting hitting vector $\vec{h}$ is a core, then the algorithm can do the usual core improvement adding $\vec{k}$ to $\mathcal{K}$ as it would have happened with either IHS-lb or IHS-ub. If $\vec{h}$

is a solution there are two cases. If its cost is less than ub, the upper bound is updated as it would have happened with IHS-ub. Alternatively, if its cost is more than or equal to ub, then there is no use for $\vec{k}$ and the iteration has been useless. To avoid the algorithm entering infinite loops, [11] suggests forcing the following iteration not to use the greedy algorithm. In our case, depending on whether the next iteration computes an optimal hitting vector or a cost-bounded hitting vector we will denote the algorithm IHS-$grdlb$ or IHS-$grdub$.

4.2 Improving Cores

The different alternatives considered so far aimed at alleviating the time spent in computing hitting vectors. Now we address the task of improving cores. Adding to $\mathcal{K}$ cores with high values in their components is beneficial because they will increase the set of cores that they dominate and they will likely contribute towards the lower bound part of the Termination Condition. However, computing high-cost cores is more time consuming, and the right trade-off must be found.

We are restricting ourselves to algorithms that depart from a hitting vector $\vec{h}$ and make a sequence of greedy increments to its components until some stopping criteria are achieved while preserving the core condition. In the following, we consider four alternatives.

- *Maximal Cores.* The strongest (and most time consuming) criterion is *core maximality* (that is, achieving a vector that cannot be increased in any of its components without losing the core condition). Our algorithm is reminiscent of the so-called *destructive MUS extraction* [20]. It uses a set I that contains the list of indices that may be increased. While the set I is not empty, an index is selected and its component is increased. If the resulting vector is not a core, the increment is undone and the indexed is removed from I. If a component cannot be further increased because it has reached its maximum cost, the indexed is also removed from I.
- *Minimal Cores.* The weakest (and least time consuming) option for improving cores is not to do any improvement of $\vec{h}$ whatsoever. However, since our implementation of *SolveCSP()* uses an assumptions-based SAT solver, we can take advantage of the core improvement that it gives us for free as part of the resolution.
- *Cost-bounded Cores.* One intermediate option is to improve cores until their cost reaches the upper bound. If the cost of $\vec{h}$ is less than the optimal value w^* and the cost of the resulting core $\vec{k}$ is more than or equal to w^*, then we know that adding $\vec{k}$ to the set of cores $\mathcal{K}$ will contribute towards the lower bound part of the TC because at least one more goal core will be dominated. Obviously, during the execution, we do not know the value of w^*, but hitting vectors computed by any of the four algorithms, at least at early iterations, are likely to have cost less than w^*. Then, if we improve them until their cost equals ub, the new set of cores will certainly have contributed towards TC.
- *Partially Maximal Cores.* Another alternative between minimal and maximal cores that was already used in [13] is to soften the condition of core maximality and request that the increment of just one component (instead of all of them) produces a solution vector.

In all our implementations of *ImprCore()*, we select at each iteration the index i with the lowest value v_i among the set of candidates. The reason is that having in $\mathcal{K}$ cores whose minimum value is as large as possible makes it more costly to be hit, which, in turn, makes the lower bound grow faster.

Problem	Variables	Max dom size	Constraints	Cost functions	Max costs
Rnd domains	16 - 23	30	44 - 59	44 - 59	6 - 9
Rnd weights	16 - 23	5	48 - 68	48 - 68	20 - 21
Rnd sparse	34 - 46	5	99 - 122	99 - 122	7 - 9
Rnd scale-free-4	23 - 25	5	92 - 99	92 - 99	7 -9
Rnd scale-free-5	24 - 25	5	107 - 114	107 - 114	8 -9
Ehi	297 - 315	7	4081 - 4400	4081 - 4400	2
SPOT5	45 - 506	4	122 - 9325	41 - 440	2 - 3
driverlog	46 - 546	4 - 12	156 - 14429	55 - 702	3 - 7
Grid	396 - 400	2	757 - 801	757 - 801	3
Normalized	101 - 8621	2	227 - 19903	57 - 195	2 - 10
Pedigree	208 - 9403	3 - 10	447 - 33795	180 - 10621	3 - 17
CELAR	13 - 222	14 - 44	65 - 944	65 - 876	11 - 137

Table 1 Summary of the benchmarks main characteristics: range of variables, largest domains, number of constraints, number of cost functions and maximum number of different costs appearing in the cost functions. Note that these values have been collected after VAC pre-processing.

5 Empirical Results

The experiments reported ran on nodes with 4 cores 16Gb *Dell PowerEdge* R240 with *Intel Xeon* E-2124 of 3.3Ghz. *MinCostHV()* and *CostBoundedHV()* were modeled as 0-1 integer programs and solved with *CPLEX* [10]. Induced CSPs were encoded as CNF SAT formulas and solved with *CaDiCaL* [6].

For the experiments, we used several benchmarks aiming at a heterogeneous sample of instances. Table 1 summarizes the features of each group of instances. All instances are pre-processed and made *virtually arc consistent* (VAC) [8]. Unless indicated otherwise, we use the *cost-function merging* formulation proposed in [18] where clusters of cost functions are heuristically determined using a tree decomposition and (virtually) merged into a single cost function. Also, our implementation may compute several *disjunctive cores* in the same iteration as proposed in [13]. All executions had a time out of one hour. We conducted the empirical evaluation over the following benchmarks:

Uniform random instances are characterized by five parameters (n, d, m, w, t) that correspond to the number of variables, domain size, number of binary cost functions, number of different weights at each cost function and number of tuples with non-zero cost at each cost function, respectively. The scope of the m (out of $\frac{n(n-1)}{2}$ alternatives) cost functions, the t (out of d^2 alternatives) tuples with non-zero cost and their actual cost (out of the w alternatives) are decided using a uniform random distribution. We generated 3 groups of instances aiming at increasing one of the parameters: *domains* $(25, 30, 50, 5, 750)$, *weights* $(25, 5, 50, 10000, 20)$, and *sparse* $(50, 5, 100, 5, 20)$. For each group we generated 50 instances.

Uniform random instances have been long used for testing purposes, but they are sometimes questioned because real instances are anything but random. A more realistic graph structure that has been used for empirical testing are *scale-free graphs* [2]. It is known that scale-free networks appear in many real-world networks like the World Wide Web, some social networks like papers co-authorship or citation, protein interaction network, *etc.* In our scale-free instances the constraint graph is a scale-free graph following the *Barabási-Albert* model. Instances are also characterized by five parameters (n, d, m, w, t) but unlike uniform random instances, m refers to the model's parameter. We report results for the classes $(25, 5, 4, 5, 20)$ and $(25, 5, 5, 5, 20)$.

Finally, we selected miscellaneous instances from the well-known *evalgm* repository[1] that were within the reach of IHS-based algorithms. The selection includes includes: *EHI* (Random 3-SAT instances embedding a small unsatisfiable part and coverted into a binary CSP), *SPOT5* (satellite scheduling), *driverlog* (planning in temporal and metric domains), *grid* (Markov Random Field), normalized (MIPLib), *pedigree* (genetic Linkage) and *CELAR* (frequency assignment).

5.1 Results

Our first analysis is about the impact of using or not using cost-function merging. Table 2 reports the relative time performance gain of doing cost-function merging. For each benchmark, speed-up is the solving time ratio of its best performing algorithm out of the 16 alternatives with and without cost-function merging. We observe that cost-function merging is consistently useful producing significant speed-ups that in some cases are over 250. The only case where cost-function merging is not advantageous is with *Grid* instances where its use causes none of the 16 algorithms to solve any instance (not even with a larger time limit of 4 hours). Interestingly, without cost-function merging IHS-ub cores can solve all the instances with maximal and cost-bounded cores. The most probable reason is that the grid structure is so regular that the tree-decompostion used to decide which functions to merge is not appropriate. Because this result is so conclusive, and for the sake of clarity, in the following every table reports results with cost-merging except for Grid where the reported results are without cost-merging.

Problem	Speed-up
Rnd domains	268.31
Rnd weights	2.18
Rnd sparse	39.33
Rnd scale-free-4	93.70
Rnd scale-free-5	12.69
Ehi	99.53
SPOT5	156.74
driverlog	1.61
Grid	0.46
Normalized	1.32
Pedigree	1.34
CELAR	4.85

Table 2 Relative time performance gain of cost-function merging. On a given benchmark, speed-up is the solving time ratio of its best performing algorithm with and without cost-function merging.

Tables 3 and 4 report for each one of the 12 problem classes and each one of the 16 algorithms, the relative performance with respect to time and space, respectively. Each table entry is the ratio w.r.t. the best-performing algorithm on that benchmark. For example, in

[1] http://genoweb.toulouse.inra.fr/ degivry/evalgm/

Table 3, a 1 identifies the fastest algorithm while a value of r indicates that the algorithm is r times slower than the best.

Our next analysis is about the impact that each algorithm has on the solving time (Table 3). Our first observation is that different algorithms have very different running times, but no algorithm dominates the others. In some benchmarks, that difference is so extreme that the best approach solves all instances while other approaches are not able to solve any of the instances within the time limit. If we look at problem classes where every instance is solved with all (or nearly all) algorithms and compare speed-up ratios, we still see that the best approach is at least 4 times faster than the worst and the different can go up to several orders of magnitude. We also observe that some benchmarks are very sensitive with respect to the method in which hitting vectors are computed (e.g. *Grid*, *scale-free-5*), some benchmarks are very sensitive to the method in which cores are improved (e.g. *CELAR*) and some benchmarks are very sensitive to both (e.g. *SPOT5*). Therefore, one should use caution before concluding from experiments that the HS approach is not suitable for a particular type of problem, because it may happen that the right algorithm has not been considered.

Regarding the different ways to compute hitting vectors, we observe that the best option is again highly benchmark dependent. The best algorithm is 5 times with IHS-grdub, 3 times with IHS-ub and 4 times with IHS-lb. IHS-grdlb never appears in the best algorithm, but when the best algorithm is IHS-lb, it usually performs very closely. We observed that in the many cases in which greedy vectors are not beneficial the reason is that except for the very first iterations, greedy hitting vectors produce cheap but useless iterations and IHS-*grdlb* (resp. IHS-*grdub*) converges to IHS-*lb* (resp. IHS-*ub*). From that we conjecture that making more effective greedy algorithms, even at the cost of being more time consuming may be a useful improvement.

Regarding the different ways to improve cores, we observe that the most common best option is to compute maximal cores. The only two exceptions are *Pedigree* and *random-domains* where the best option is to compute minimal cores. The reason for the exception is that in these two benchmarks the induced CSPs are very difficult for the SAT solver and it pays off to generate much larger sets of cores even if it is at the cost of making the computation of hitting vectors more difficult. This observation let us believe that a promising improvement for this type of instances would be to find more efficient SAT solvers or even switching to some other solving paradigm.

It is surprising that the two extreme core improvement methods (maximal and minimal cores) are best options and the intermediate methods never are so. Inspecting the results with more detail we observe that computing cost-bounded cores is often equivalent to not making any core improvement because the core that the assumption-based SAT solver provides for free already has a cost larger than the *ub*. We also observed that the cost of partially maximal cores is often too close to the cost of minimal cores (for example in *normalized*), making them to weak.

Our third analysis is about the impact that each algorithm has on the number of cores $|\mathcal{K}|$ needed to achieve the Termination Condition $\mathcal{GC} \leq \mathcal{K}$ (Table 4). The final size of $\mathcal{K}$ is relevant because it is related to both the number of iterations and the space requirements.

Regarding hitting vector computation, the pattern is clear. As expected, IHS-*lb* (optimal hitting vectors) requires fewer cores than IHS-*ub* cores to dominate the set of goal cores, but the difference does not seem to be dramatic. The same thing happens when considering greedy hitting vectors. From that, we conjecture that cost-bounded cores quickly become nearly as good as optimal cores (probably because the *ub* gets tight) and the cost of greedy cores quickly becomes too high and therefore they become useless.

Regarding core improvement, the pattern is also clear. As expected, the weaker the improvement, the more cores are needed. Compared with maximal cores, the use of partially maximally maximal cores needs a larger $|\mathcal{K}|$ but the difference usually is not very large. When considering cost-bounded cores the set $\mathcal{K}$ ends up being much larger (e.g. *Ehi*), which means that the effort of improving cores until their cost reaches the upper bound is not enough in general to dominate more goal cores and accelerate the termination of the execution. As a matter of fact, using cost-bounded cores has an effect very close to using minimal cores. A probable reason is that the cores provided by the assumption-based SAT solver often satisfy that their cost reaches the ub, so both approaches end up being equivalent.

From the experiments presented in this paper (and many others that we have also conducted), we can say that the effectiveness of IHS algorithms still falls far behind the state-of-the-art *Toulbar2* solver [17]. However, in our experiments we see that there are instances where IHS is competitive. For instance, Table 5 compares average solving times and number of unsolved instances with Toulbar2 (version 1.2) and the best IHS among all the alternatives that we considered. We can see that IHS outperforms toulbar2 in Ehi, SPOT5, Grid, Normalized and Pedigree, but toulbar2 outperforms IHS in all the random benchmarks. All the instances where IHS seems suitable have in common small domains and not too many different costs in their cost functions. Small domains are good for solving induced CSPs encoding and solving them with SAT. Not too many different costs means smaller vector spaces. The conjunction of these two features seems a very reasonable proxy for IHS suitability and designing improvements for problems having this features may be the fastest route to find regions where IHS may be competitive.

	grdub	ub	grdlb	lb	grdub	ub	grdlb	lb
	SPOT5 (13)				Pedigree (21)			
minimal	25.50 (1)	28.83 (1)	41.73 (0)	82.29 (2)	1 (0)	3.91 (0)	3.72 (1)	4.07 (2)
cost-bounded	27.26 (1)	29.19 (1)	52.99 (0)	80.58 (3)	3.34 (0)	3.63 (0)	3.56 (1)	3.71 (2)
partially maximal	22.45 (1)	22.31 (1)	7.62 (0)	7.62 (0)	3.52 (0)	3.62 (0)	3.68 (0)	3.58 (1)
maximal	21.82 (1)	21.74 (1)	1.97 (0)	1 (0)	3.52 (0)	3.37 (0)	3.47 (0)	3.26 (0)
	driverlog (23)				CELAR (9)			
minimal	217.00 (6)	298.69 (8)	296.86 (6)	408.44 (12)	7.52 (4)	11.29 (7)	8.41 (5)	11.29 (7)
cost-bounded	210.87 (4)	285.06 (7)	293.31 (7)	414.61 (13)	7.68 (4)	11.29 (7)	8.46 (5)	7.73 (4)
partially maximal	56.86 (1)	54.70 (1)	62.74 (1)	99.32 (1)	1.86 (1)	4.95 (3)	1.82 (1)	5.04 (3)
maximal	1.25 (0)	1.23 (0)	1.36 (0)	1 (0)	1.78 (1)	1.83 (1)	1.72 (1)	1 (0)
	Grid (5)				Ehi (200)			
minimal	2.18 (5)	1.30 (1)	2.18 (5)	2.18 (5)	14.34 (0)	21.65 (0)	21.16 (0)	114.02 (13)
cost-bounded	2.18 (5)	1.42 (0)	2.18 (5)	2.18 (5)	14.44 (0)	21.40 (0)	21.03 (0)	113.56 (10)
partially maximal	2.18 (5)	1.05 (1)	2.18 (5)	2.18 (5)	1.41 (0)	1.28 (0)	1.48 (0)	1.51 (0)
maximal	2.18 (5)	1 (0)	2.18 (5)	2.18 (5)	1.02 (0)	1 (0)	1.10 (0)	1.04 (0)
	Normalized (4)				Rnd scale-free-4 (20)			
minimal	2.56 (2)	2.82 (2)	2.56 (2)	3.82 (3)	90.77 (18)	91.97 (18)	93.64 (19)	98.50 (20)
cost-bounded	2.56 (2)	2.82 (2)	2.56 (2)	3.82 (3)	90.23 (18)	91.09 (18)	93.63 (19)	98.50 (20)
partially maximal	2.55 (2)	2.55 (2)	2.55 (2)	2.56 (2)	37.04 (3)	34.77 (3)	81.75 (14)	89.33 (17)
maximal	1 (0)	1.44 (0)	2.32 (1)	2.55 (2)	1.24 (0)	1 (0)	16.39 (1)	19.18 (1)
	Rnd scale-free-5 (20)				Rnd weights (50)			
minimal	12.69 (20)	12.69 (20)	12.69 (20)	12.69 (20)	4.09 (49)	4.11 (50)	4.01 (48)	4.06 (49)
cost-bounded	12.69 (20)	12.69 (20)	12.69 (20)	12.69 (20)	4.11 (50)	4.11 (50)	4.01 (47)	4.06 (49)
partially maximal	8.94 (11)	8.66 (11)	12.11 (19)	12.23 (19)	3.65 (41)	3.68 (41)	2.80 (27)	2.84 (29)
maximal	1 (0)	1.01 (0)	7.83 (9)	7.68 (9)	2.63 (21)	3.17 (28)	1.34 (4)	1 (1)
	Rnd domains (50)				Rnd sparse (50)			
minimal	1 (0)	3.28 (0)	1.28 (0)	4.35 (0)	75.91 (19)	97.88 (24)	82.13 (20)	143.50 (37)
cost-bounded	1 (0)	1.35 (0)	1.11 (0)	2.58 (0)	73.87 (16)	98.75 (23)	80.05 (20)	143.88 (36)
partially maximal	3.35 (0)	1.30 (0)	3.41 (0)	2.31 (0)	22.89 (3)	34.39 (5)	40.11 (7)	98.25 (24)
maximal	3.80 (0)	2.60 (0)	3.95 (0)	3.12 (0)	1 (0)	2.18 (0)	6.08 (0)	21.25 (2)

Table 3 Relative time performance of 16 different IHS algorithms on 12 different problem classes using cost-function merging (except for the Grid class). For each class (in parenthesis its number of instances), columns indicate the method to compute hitting vectors and rows indicate the method to improve them. Each entry is the ratio of its average solving time over the average solving time of the best performing alternative in that benchmark (in parenthesis the number of unsolved instances within the time limit of 1 hour).

6 Conclusion and Future Work

We have presented a large empirical evaluation of 32 alternative implementations of the Implicit Hitting Set approach for WCSP solving. Although our current implementations of IHS are only competitive with the state-of-the-art Toulbar2 solver in selected instances, our results show how different is the performance of different alternatives, and we believe that it indicates that the approach is very general and has potential.

We covered a variety of alternatives, but many known improvements that have been found useful in other paradigms remain to be adapted to the WCSP framework and tested. For example, we want to consider in the future reduced cost fixing or weight-aware cost extraction [23]. It is reasonable to expect that the IHS will also benefit from them in the WCSP paradigm.

More importantly, we believe that all the components of the algorithms that we have tested can be improved. We want to evaluate alternative ways to solve induced CSPs and a natural option is to replace the SAT solver by a constraint programming solver. We also want to evaluate alternative ways to find cost-bounded hitting vectors and a natural option would be to replace CPLEX by a Pseudo-Boolean optimization solver or a SAT solver with one of the many efficient encodings of Pseudo-Boolean constraints. Finally, we want to evaluate alternatives to the greedy algorithm and we believe that local search is a promising direction.

References

1 David Allouche, Simon de Givry, George Katsirelos, Thomas Schiex, and Matthias Zytnicki. Anytime hybrid best-first search with tree decomposition for weighted CSP. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 12–29. Springer, 2015. doi:10.1007/978-3-319-23219-5_2.

2 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Scale-free random SAT instances. *Algorithms*, 15(6):219, 2022. URL: https://doi.org/10.3390/a15060219, doi:10.3390/A15060219.

3 Abdelkader Beldjilali, Pierre Montalbano, David Allouche, George Katsirelos, and Simon de Givry. Parallel hybrid best-first search. In Christine Solnon, editor, *28th International Conference on Principles and Practice of Constraint Programming, CP 2022, July 31 to August 8, 2022, Haifa, Israel*, volume 235 of *LIPIcs*, pages 7:1–7:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.CP.2022.7, doi:10.4230/LIPICS.CP.2022.7.

4 E. Bensana, Michel Lemaître, and Gérard Verfaillie. Earth observation satellite management. *Constraints An Int. J.*, 4(3):293–299, 1999. doi:10.1023/A:1026488509554.

5 Jeremias Berg, Fahiem Bacchus, and Alex Poole. Abstract cores in implicit hitting set maxsat solving. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 277–294. Springer, 2020. doi:10.1007/978-3-030-51825-7_20.

6 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

7 Bertrand Cabon, Simon de Givry, Lionel Lobjois, Thomas Schiex, and Joost P. Warners. Radio link frequency assignment. *Constraints An Int. J.*, 4(1):79–89, 1999. doi:10.1023/A:1009812409930.

8 Martin C. Cooper, Simon de Givry, Martí Sánchez-Fibla, Thomas Schiex, Matthias Zytnicki, and Tomás Werner. Soft arc consistency revisited. *Artif. Intell.*, 174(7-8):449–478, 2010. doi:10.1016/j.artint.2010.02.001.

9 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001. URL: http://www.amazon.com/Introduction-Algorithms-Thomas-H-Cormen/dp/0262032937%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0262032937.

10 IBM ILOG Cplex. V12. 1: User's manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.

11 Jessica Davies. *Solving MAXSAT by Decoupling Optimization and Satisfaction*. PhD thesis, University of Toronto, Canada, 2014. URL: http://hdl.handle.net/1807/43539.

12 Jessica Davies and Fahiem Bacchus. Postponing optimization to speed up MAXSAT solving. In Christian Schulte, editor, *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2013. doi:10.1007/978-3-642-40627-0_21.

13 Erin Delisle and Fahiem Bacchus. Solving weighted csps by successive relaxations. In Christian Schulte, editor, *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 273–281. Springer, 2013. doi:10.1007/978-3-642-40627-0_23.

14 Katalin Fazekas, Fahiem Bacchus, and Armin Biere. Implicit hitting set algorithms for maximum satisfiability modulo theories. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2018. doi:10.1007/978-3-319-94205-6_10.

15 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

16 Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 329–376. Elsevier, 2006. doi:10.1016/S1574-6526(06)80014-3.

17 Barry Hurley, Barry O'Sullivan, David Allouche, George Katsirelos, Thomas Schiex, Matthias Zytnicki, and Simon de Givry. Multi-language evaluation of exact solvers in graphical model discrete optimization. *Constraints An Int. J.*, 21(3):413–434, 2016. doi:10.1007/s10601-016-9245-y.

18 Javier Larrosa, Conrado Martinez, and Emma Rollon. Theoretical and empirical analysis of cost-function merging for implicit hitting set wcsp solving. In *The 38th Annual AAAI Conference on Artificial Intelligence February 20-27, 2024; Vancouver, Canada*. AAAI Press, 2024.

19 Javier Larrosa and Thomas Schiex. Solving weighted CSP by maintaining arc consistency. *Artif. Intell.*, 159(1-2):1–26, 2004. doi:10.1016/j.artint.2004.05.004.

20 João Marques-Silva and Inês Lynce. On improving MUS extraction algorithms. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2011. doi:10.1007/978-3-642-21581-0_14.

21 Paul Saikko, Carmine Dodaro, Mario Alviano, and Matti Järvisalo. A hybrid approach to optimization in answer set programming. In Michael Thielscher, Francesca Toni, and Frank Wolter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the*

Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018, pages 32–41. AAAI Press, 2018. URL: https://aaai.org/ocs/index.php/KR/KR18/paper/view/18021.

22 Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. Pseudo-boolean optimization by implicit hitting sets. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 51:1–51:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL: https://doi.org/10.4230/LIPIcs.CP.2021.51, doi:10.4230/LIPICS.CP.2021.51.

23 Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. Improvements to the implicit hitting set approach to pseudo-boolean optimization. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 13:1–13:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.SAT.2022.13, doi:10.4230/LIPICS.SAT.2022.13.

24 Clément Viricel, Simon de Givry, Thomas Schiex, and Sophie Barbe. Cost function network-based design of protein-protein interactions: predicting changes in binding affinity. *Bioinform.*, 34(15):2581–2589, 2018. doi:10.1093/bioinformatics/bty092.

25 Jelena Vucinic, David Simoncini, Manon Ruffini, Sophie Barbe, and Thomas Schiex. Positive multistate protein design. *Bioinform.*, 36(1):122–130, 2020. URL: https://doi.org/10.1093/bioinformatics/btz497, doi:10.1093/BIOINFORMATICS/BTZ497.

	grdub	ub	grdlb	lb	grdub	ub	grdlb	lb
	SPOT5				Pedigree			
minimal	6.23	6.33	5.94	6.74	1.55	1.36	1.50	1.27
cost-bounded	6.46	6.80	5.68	6.11	1.14	1.17	1.17	1.15
partially maximal	3.04	2.35	2.73	2.24	1.25	1.17	1.23	1.10
maximal	1.67	1.57	1.27	1	1.13	1.05	1.09	1
	driverlog				CELAR			
minimal	31.80	40.77	23.24	21.30	86.75	32.23	80.09	8.90
cost-bounded	29.96	39.08	22.46	21.90	85.75	31.17	81.91	5.23
partially maximal	10.17	8.67	9.66	8.49	20.91	9.27	15.87	3.44
maximal	2.08	1.25	2.00	1	6.27	3.68	5.36	1
	Grid				Ehi			
minimal	87.18	6.61	87.27	1.06	282.08	181.58	282.00	290.17
cost-bounded	69.33	6.61	86.72	1.05	282.53	181.98	281.83	290.82
partially maximal	86.93	7.88	86.90	1.04	7.05	4.95	7.03	5.84
maximal	84.41	7.61	84.55	1	1.16	1	1.19	1.10
	Normalized				Rnd scale-free-4			
minimal	10.95	5.05	6.27	1.93	23.34	21.93	7.63	4.68
cost-bounded	11.41	5.03	5.66	1.93	24.21	22.19	7.76	4.58
partially maximal	6.28	3.64	3.62	1.16	7.91	7.55	3.81	3.04
maximal	2.43	1.49	1.97	1	1.48	1.24	1.29	1
	Rnd scale-free-5				Rnd weights			
minimal	15.38	15.62	3.56	3.37	17.35	1.63	33.79	16.88
cost-bounded	15.16	15.86	3.35	3.32	19.91	1.69	37.05	16.61
partially maximal	9.68	9.49	2.11	1.95	4.29	1.47	10.89	8.38
maximal	2.07	2.06	1.18	1	2.28	1	2.58	1.38
	Rnd domains				Rnd sparse			
minimal	15.29	8.87	15.84	9.43	95.52	15.14	89.26	4.22
cost-bounded	10.69	5.30	10.79	5.86	95.48	15.12	83.46	4.15
partially maximal	8.02	2.28	7.80	3.38	20.97	6.60	21.50	2.81
maximal	1.55	1	1.58	1.07	2.67	1.69	2.56	1

Table 4 Relative space performance (measured as the size of the set of cores at termination or at time out) of 16 different IHS algorithms on 12 different problem classes using cost-function merging. Each entry is the ratio average $|\mathcal{K}|$ divided by the minimum $|\mathcal{K}|$ among the algorithms.

Problem	**Toulbar2**		**IHS (best)**	
	time	**nb. unsolved**	**time**	**nb. unsolved**
Rnd domains	0.65	0	13.42	0
Rnd weights	0.01	0	509.04	1
Rnd sparse	0.02	0	20.15	0
Rnd scale-free-4	0.01	0	36.55	0
Rnd scale-free-5	0.03	0	283.73	0
Ehi	136.57	1	10.86	0
SPOT5	2367.06	8	13.25	0
driverlog	0.45	0	3.80	0
Grid	3600	5	1652.77	0
Normalized	912.42	1	707.93	0
Pedigree	631.39	3	81.96	0
CELAR	0.46	0	248.06	0

Table 5 Average running times (in seconds) and number of unsolved instances within the time limit of 1 hour. Toulbar2 is executed with default parameters. For each benchmark, IHS is its best among all the alternatives.