An Adaptive Implicit Hitting Set Algorithm for MAP and MPE Inference

Aleksandra Petrova APETROVA@CS.UPC.EDU
Javier Larrosa LARROSA@CS.UPC.EDU
Emma Rollon EROLLON@CS.UPC.EDU
Universitat Politècnica de Catalunya, Barcelona, Spain

Editors: J.H.P. Kwisthout & S. Renooij

Abstract

In this paper, we address the use of the implicit hitting set approach (HS) for MAP (Markov Random Fields) and MPE (Bayesian Networks). Since the HS approach is quite general and finding the best version is very problem-dependent, here we present an adaptive algorithm that learns a reasonably good version for the instance being solved. The algorithm, which follows a Multi-armed Bandit structure, explores the different alternatives as it iterates and adapts their weights based on their performance. The weight is used to decide on the probability of selecting a given alternative in the next iteration.

Keywords: MAP and MPE inference; Implicit Hitting Set Approach; Multi-armed Bandit.

1. Introduction

Probabilistic Graphical Models (PGM) such as *Markov Random Fields* (MRF) or *Bayesian Networks* (BN) have been successfully used to model and solve a variety of problems in image analysis, genetics, structural biology, and bioinformatics Koller and Friedman (2009). Here we are concerned with one of the essential queries on MRF and BN which is to find an optimal assignment to all variables. For MRF this problem is known as *Maximum A Posteriori* (MAP-MRF). For BN it is known as *Maximum Probability Explanation* (MPE-BN). We consider both MRF and BN under the unifying framework of *Cost Networks* where potentials and conditional probability tables are log-transformed and either MRF-MAP or BN-MPE correspond to finding an optimal assignment in the associated Cost Network. This problem is usually known as the *Weighted Constraint Satisfaction Problem* (WCSP) Allouche et al. (2015).

The topic of this paper is the application of the recently proposed Implicit Hitting Set (HS) approach for WCSP. Roughly speaking, it consists of a loop such that at each iteration an unsatisfiable piece of the problem (a.k.a. core) is uncovered. From the set of cores, the algorithm computes an anytime lower bound of the optimum. From the process of growing cores (i.e., increasing its components), the algorithm computes an anytime upper bound of the optimum. The algorithm stops when both bounds meet. Note that this is an exact approach, meaning that it returns the true optimum rather than an approximation. Since the problem is NP-hard, it follows that the algorithm is worst-case time exponential.

Although the HS approach has been found effective in a variety of related discrete optimization settings (Davies, 2014; Berg et al., 2020; Smirnov et al., 2021, 2022; Saikko et al., 2018), its efficient application for WCSP remains an open challenge (Delisle and

Bacchus, 2013; Larrosa et al., 2024). One major difficulty is that HS is a relatively generic framework. There are several ways to implement the body of the loop and different options may have very different performances. Furthermore, there does not seem to be a dominant option. On the contrary, the best option is highly domain-dependent.

The main contribution of this paper is a Multi-armed Bandit implementation of HS such that each arm corresponds to one possible loop alternative. To the best of our knowledge, no adaptive version of HS has been tried before. We observe on a number of benchmarks that our approach consistently outperforms the average of the different alternatives which, we believe, shows its promise.

2. Background

A *graphical model* (Dechter (2019)) is a tuple $(X, F, \oplus)$ where X are *variables* each one taking values on a discrete *domain*. F is a set of *local functions*. Each function is defined on a subset of the variables called its *scope* and maps each possible assignment to a value. The use of the $\oplus$ operator (called combinator) is to *combine* the values of the local functions. The graphical model represents a *global function* whose scope is X which is the combination of all its local functions $\bigoplus_{f \in F}^{m} f$

In this paper, we are concerned with two types of graphical models: Constraint Networks and Cost Networks. In *constraint networks* the local functions are *constraints* (i.e., functions that assign a Boolean value to any assignment in its domain) and the combination operator is the boolean *and* $\wedge$. The most usual task is called the *Constraint Satisfaction Problem* (CSP) and consists of finding an assignment of the variables that satisfies all the constraints.

In *Cost Networks* the local functions are *cost-functions* (i.e, functions that assign a natural number to any assignment in its domain) and the combination operator is the sum $+$ (i.e, the global function is the sum of local costs). The most usual task is called *Weighted* CSP (WCSP) and it consists of finding an assignment of the variables that minimizes the sum of costs,

$$\min_{X} \sum_{f \in F} f(\cdot)$$

It is well-known that Markov Random Fields and Bayesian Networks can be represented as Cost Networks where the cost functions are log-transformations of potential and CPTs, respectively Koller and Friedman (2009). In that case, the WCSP task corresponds to MRF-MAP and BN-MPE, respectively.

As we will see next, HS algorithms solve WCSP problems by solving a sequence of CSP problems.

3. Implicit Hitting Set

In this section, we briefly review the HS approach for WCSP solving. A more detailed description can be found in Larrosa et al. (2024). Consider an arbitrary Cost Network $(X, F, +)$ with m cost functions $F = \{f_1, f_2, \ldots, f_m\}$. Remember that our goal is to obtain the cost-optimal solution noted w^*. A *cost vector* $\boldsymbol{v} = (v_1, v_2, \ldots, v_m)$ 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. Given a set of vectors $\mathcal{K}$, a *Hitting Vector* $\boldsymbol{h}$ is a vector such that for each $\boldsymbol{v} \in \mathcal{K}$ the vector $\boldsymbol{h}$ is higher in at least one of its components. The *Minimum Cost Hitting Vector* (noted $MHV(\mathcal{K})$) is a hitting vector of minimum cost where $cost(\boldsymbol{h}) = \sum_i h_i$

Given a Cost Network, a cost vector $\boldsymbol{v}$ *induces* a Constraint Network $(X, F_{\boldsymbol{v}}, \wedge)$ where $F_{\boldsymbol{v}}$ 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 $\boldsymbol{v}$ is satisfiable we will say that $\boldsymbol{v}$ is a *solution vector*. Otherwise, we will say that $\boldsymbol{v}$ is a *core vector*.

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

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

HS algorithms aim at finding a solution vector $\boldsymbol{h}$ and a set of cores $\mathcal{K}$ such that the two bounds meet (that is, $MHV(\mathcal{K}) = cost(\boldsymbol{h})$). In the following, we enumerate four different alternative ways to achieve this goal. The first one was first proposed in Delisle and Bacchus (2013) and the other three have been adapted from MaxSAT and Pseudo-boolean Optimization. All the algorithms consist of one loop whose body can be divided into two parts. Firstly, they find a hitting vector $\boldsymbol{h}$. Secondly, if it is a core, they obtain a larger[1] (namely, improved) one $\boldsymbol{k}$ by increasing appropriately its components while preserving its unsatisfiability.

1. HS-A (Algorithm 1 left). In this alternative, a minimum cost hitting vector *MHV()* is computed. If it is a core it is improved. Otherwise, it is known that $\boldsymbol{h}$ is an optimal solution and the algorithm can stop.

2. HS-B (Algorithm 1 right). In this alternative, a near-optimal hitting vector is computed using a greedy algorithm $HVgreedy()$. If it is a core, then it is improved. Else, if it is a solution the upper bound is possibly updated. If the upper bound is not updated, then the iteration has been wasted, which means that this algorithm is not complete since it can iterate infinitely many times. This problem can be solved by forcing the algorithm to iterate as in the other alternatives once in a while Davies (2014).

3. HS-C (Algorithm 2 left). In this alternative, a hitting vector with a cost less than the upper bound is computed. The idea is to emulate HS-B but disallow wasted iterations.

4. HS-D (Algorithm 2 right). This alternative is similar to the previous one, but now trying to behave closer to HS-A by forcing the hitting vector to be lower (that is, closer to the optimum)

All four variations may follow different strategies for growing (namely, improving) the core. The two possibilities that are explored in this paper are: *Maximal cores* and *Partially maximal cores*. With Maximal cores, we achieve a cost vector $\boldsymbol{h}$ which cannot be increased

1. Recall the usual partial order among vectors where $\boldsymbol{a} < \boldsymbol{b}$ if $\boldsymbol{a} \neq \boldsymbol{b}$ and $\boldsymbol{b}$ is larger than or equal to $\boldsymbol{a}$ in every component

```
Function HS-A($X, C, F$)
begin
    $\mathcal{K} := \emptyset$;  $lb := 0$;  $ub := \infty$
    while $lb < ub$ do
        $\boldsymbol{h}$ :=MHV($\mathcal{K}$);
        $lb := cost(\boldsymbol{h})$;
        if SolveCSP($X, C \cup F_{\boldsymbol{h}}$) then
            $ub := cost(\boldsymbol{h})$
        end
        else
            $\boldsymbol{k}$ :=growCore($X, C, F, ub, \boldsymbol{h}$);
            $\mathcal{K} := \mathcal{K} \cup \{\boldsymbol{k}\}$;
        end
    end
    return $lb$
end
```

```
Function HS-B($X, C, F$)
begin
    $\mathcal{K} := \emptyset$;  $lb := 0$;  $ub := \infty$
    while $lb < ub$ do
        $\boldsymbol{h}$ :=HVgreedy($\mathcal{K}$);
        if SolveCSP($X, C \cup F_{\boldsymbol{h}}$) then
            $ub := min(cost(\boldsymbol{h}), ub)$;
        end
        else
            $\boldsymbol{k}$ :=growCore($X, C, F, ub, \boldsymbol{h}$);
            $\mathcal{K} := \mathcal{K} \cup \{\boldsymbol{k}\}$;
        end
    end
    return $lb$
end
```

Algorithm 1: Two alternative HS algorithms for solving WCSP. They receive as input a WCSP (X, C, F) and return the cost of the optimal solution w^*. Function *growCore()* receives as input a core $\boldsymbol{h}$ and returns a core $\boldsymbol{k}$ such that $\boldsymbol{h} \leq \boldsymbol{k}$.

```
Function HS-C($X, C, F$)
begin
    $\mathcal{K} := \emptyset$;  $lb := 0$;  $ub := \infty$
    while $lb < ub$ do
        $\boldsymbol{h}$ :=HV($\mathcal{K}, ub$);
        if $\boldsymbol{h} = NUL$ then $lb := ub$;
        else
            if SolveCSP($X, C \cup F_{\boldsymbol{h}}$) then
                $ub := cost(\boldsymbol{h})$;
            end
            else
                $\boldsymbol{k}$ :=growCore($X, C, F, ub, \boldsymbol{h}$);
                $\mathcal{K} := \mathcal{K} \cup \{\boldsymbol{k}\}$;
            end
        end
    end
    return $lb$
end
```

```
Function HS-D($X, C, F$)
begin
    $\mathcal{K} := \emptyset$;  $lb := 0$;  $ub := \infty$
    while $lb < ub$ do
        $\boldsymbol{h}$ :=HV($\mathcal{K}, \frac{lb+ub}{2}$);
        if $\boldsymbol{h} = NUL$ then $lb := \frac{lb+ub}{2}$;
        else
            if SolveCSP($X, C \cup F_{\boldsymbol{h}}$) then
                $ub := cost(\boldsymbol{h})$;
            end
            else
                $\boldsymbol{k}$ :=growCore($X, C, F, ub, \boldsymbol{h}$);
                $\mathcal{K} := \mathcal{K} \cup \{\boldsymbol{k}\}$;
            end
        end
    end
    return $lb$
end
```

Algorithm 2: Two additional alternative HS algorithms for solving WCSP.

in any of its dimensions without losing the core condition. Our implementation is inspired in Marques-Silva and Lynce (2011), where through a list of dimensions that can be increased, we select a dimension and increase it until the vector no longer is a core, which then gets reduced by 1. This is repeated until there are no dimensions to increase. The second possibility for growing cores is to increase components until at least in one dimension an increment makes the vector a solution vector Delisle and Bacchus (2013).

By combining the four loop alternatives and the two ways to grow cores we obtain eight different algorithms. We have seen in our experiments that there is no winning strategy and the performance difference between the best and the worst alternative can be dramatic. This is what motivates our adaptative approach, explained next.

4. Multi-armed Bandit

A multi-armed bandit (MAB) problem is defined by a set of slot machines (or arms), each providing a random reward from an associated unknown probability distribution. A solution or strategy decides which arm to play at each step. This solution has as a goal to balance the exploration and exploitation to maximize cumulative rewards over time. Initially, when the Reinforcement Learning agent is faced with the problem it does not have any knowledge of the probabilities. It is through the process of exploration that it can slowly learn them, so then the best ones can be exploited, providing high rewards Sutton and Barto (1998) Slivkins (2019).

Many strategies exist for learning the distribution of arms. Each strategy follows a similar framework shown in 3. At every step, the agent selects an arm a to be pulled which is based on a decision to either explore or exploit. The action of the selected arm a is then executed, upon which a reward r is received. This reward often depends on aspects of the arms that are unique to the problem that is being solved. Using the reward obtained, the agent updates the policy $N[a]$, $Q[a]$ which influences the selection of arms in the next iteration. The policy often consists of aiming to minimize the regret, the difference between the reward obtained and the reward that could have been $r - Q[a]$. The past choices $Q[a]$ of the agent can also be used to inform the policy. The policy should maximize the accumulated reward over the horizon. Several strategies have been considered for MAB, including ϵ-greedy, Softmax, or Upper Confidence Bound (UCB).

```
Function MAB($n, T$)
begin
    $Q \leftarrow$ array of zeros of size $n$
    $N \leftarrow$ array of zeros of size $n$
    for $t \leftarrow 1$ to $T$ do
        $a \leftarrow$ SelectArm($Q, N, t$) // e.g., $\epsilon$-greedy, Softmax, UCB
        $r \leftarrow$ PullArmAndGetReward($a$)
        $N[a] \leftarrow N[a] + 1$
        $Q[a] \leftarrow Q[a] + \frac{1}{N[a]}(r - Q[a])$
    end
    return $Q, N$
end
```

Algorithm 3: Multi-armed Bandit Framework. Where n represents the number of arms, T is the number of iterations, Q is the estimated value of each arm, and N counts for the selection of each arm.

5. Multi-armed Hitting Set

We formulate a Multi-armed Bandit Hitting Set for solving WCSPs with *eight arms*. The given arms are created by combining previously discussed alternatives *A-D*, with *Maximal cores* and *Partially maximal cores*. Each combination represents a unique arm that employs a different strategy to reach the goal of the problem. At each iteration of the algorithm, the agent selects an arm to pull, based on the Roulette Wheel strategy. The Roulette Wheel is an MAB algorithm that selects an action (arm) based on the probabilistic value estimate of each arm. The selection is made by picking first a random value in the range of (0,1) and cumulatively augmenting the sum of the values of the arms (based on the estimates) until we reach the randomly picked value. Once the arm has been picked, the agent receives a reward which is then used to update the estimates for each arm. This process allows for a good balance between exploration and exploitation, as the arms with higher probabilities get better odds of being picked.

There are two main motivations as to why we picked the *Roulette Wheel* as our MAB agent. Firstly, the alternatives of reaching our goal which also correspond to the arms that we are using in the algorithm are not all equal when it comes to reaching the optimum. In some of the alternatives, the algorithm can find itself looping for an infinite amount of time, while in others due to how we increase the lower bound we cannot reach the case where it is equal to the upper bound. Secondly, as previously stated there is no dominant option among the alternatives, so if an agent were to converge too slowly to one alternative, it would have a hard time ensuring efficiency. This is because once the agent has converged, the policy takes some time to reduce the estimate of the winning arm and turn to another option. Thus with the Roulette Wheel, the agent learns the *probabilities* of each arm as it progresses through the algorithm reaching the optimum, allowing for switches to still happen even if one arm is a dominant strategy.

At each iteration, the agent selects an arm which it then utilizes to obtain a cost vector $\boldsymbol{h}$, update the bounds, and grow it into cores in case the cost vector is not a solution. The selection of each arm is done by computing the probabilities of each of the eight arms and then following the aforementioned procedure of the Roulette Wheel. For the agent to know the given probabilities and select the best arm given them, we need to provide it with *rewards*. For this, we define a reward function,

$$r = \frac{(|K| + \sum_{k \in K} cost(\boldsymbol{k}) - cost(\boldsymbol{h}))}{T}$$

where K is the set of cores that the selected arm produced in the iteration, $\boldsymbol{h}$ is the hitting vector obtained in the iteration, and T is the time taken by the arm in seconds. This reward encourages the agent to select arms that have been successfully producing cores, which have as well a bigger difference in the cost from the initial cost vector, forcing the algorithm to explore unreached regions.

To complete the agent we add the third element, the *policy update*. The policy update is the formula used for the probabilities to be updated upon receiving a reward. We employ the following formulas:

$$N[a] \leftarrow N[a] + 1$$

$$Q[a] \leftarrow Q[a] + \frac{1}{N[a]}(r - Q[a])$$

. Through this policy, we prioritize the importance of the past decisions the algorithm has taken relying on the overall performance of each arm, rather than the immediate reward. The rewards are always positive, resulting in positive estimates. The final step before selecting the arm is to turn the estimates into probabilities. For this, we use the simple cumulative sum:

$$P_i = \frac{Q[i]}{\sum_{j=1}^{n} Q[j]}$$

6. Empirical Results

We conducted an assessment of our proposal by creating a benchmark of various WCSP problems and utilizing them to evaluate our proposal. This benchmark consists of *random uniform problems*, *random scale-free problems*, *random grid problems*, and *pedigree* instances. Random uniform problems are characterized by 5 parameters: the number of variables (n), domain size (d), number of binary cost functions (m), number of different weights at each cost function (w), and number of tuples with the non-zero cost at each cost function (t). The scope of cost functions and the actual cost of tuples are randomly decided using a uniform distribution. In total, we created 9 groups with different parameter combinations looking for a variety of problems with a reasonable level of difficulty. Each group contains 20 or 50 instances. Random uniform problems are often criticized because they lack the structure of real problems. To test our approach on more realistic types of problems we generated *scale-free* networks using the Barabási-Albert model Ansótegui et al. (2022). Scale-free graphs have been found in a number of real situations. Our instances are also defined in terms of five parameters (n, d, m, w, t), with n and m being the two parameters of the Barabási-Albert model and d, w, and t being the same as in the uniform random instances. 4 groups of problems were created, each with 20 instances. Aiming at even more structure, we generated random problems with the scope of cost functions forming a $n \times n$ grid. Once again the instances are created using parameters: dimension of the grid n, domain size (d), number of different weights at each cost function (w), and number of tuples with the non-zero cost at each cost function (t), similarly to the random ones. In total, we have 2 groups of 20 instances each. Our last set of problems are pedigree instances from the genetic Linkage problem taken from a standard benchmark Allouche et al. (2015).

All of the instances are pre-processed using *virtually arc consistent* (VAC) Cooper et al. (2010). In our implementation, we used *cost-function merging* as proposed in Larrosa et al. (2024), which (virtually) merges the clusters of cost functions through a tree decomposition into a single cost function. The implementation may compute some *disjunctive cores* in the same iteration as seen in Delisle and Bacchus (2013). The experiments reported ran on nodes with 4 cores 16Gb Dell PowerEdge R240 with Intel193 Xeon E-2124 of 3.3Ghz. MHV() and HV() were modeled as 0-1 integer programs and solved with CPLEX Cplex (2009). Induced CSPs were encoded as CNF SAT formulas and solved with CaDiCaL Biere et al. (2020). Each execution had a time out of 1 hour.

Problem Class	**Best alt.**	**HS (best)**	**HS (avg.)**	**HS (worst)**	**MAB**
Random-15-35-70-700-7	C with 2	237.65	484.87	613.85	438.33
Random-25-30-50-750-5	C with 2	17.43	39.97	53.01	34.77
Random-25-5-50-20-1000	A with 1	875.00	2309.33	3219.27	1439.53
Random-30-8-100-32-150	A with 1	772.37	2050.04	3003.98	862.45
Random-35-6-75-30-12	B/C with 1	144.23	1559.66	3435.64	565.71
Random-40-3-150-3-5	B/C with 1	38.14	581.14	1663.92	104.11
Random-50-5-100-20-5	B/C with 1	20.15	569.65	1979.94	170.99
Random-70-4-175-12-4	B/C with 1	1777.80	2814.29	3600	2802.14
Random-100-4-250-6-15	B with 1	15.85	520.78	1711.57	41.05
Scale-free-4-25-5-20-5	C with 1	36.55	1282.37	3264.93	300.53
Scale-free-4-50-6-15-10	B/C with 1	2045.26	3014.28	3600	2390.68
Scale-free-5-25-5-20-5	B/C with 1	283.73	2108.85	3471.02	1555.46
Scale-free-7-20-3-7-10	C with 1	1.66	33.19	138.19	4.24
Grid-30-4-8-5	B with 1	155.19	843.93	1860.37	450.40
Grid-50-3-5-5	B with 1	93.36	519.26	1192.53	199.85
Pedigree	B/C with 1	27.81	101.08	253.16	71.19

Table 1: Performance of the different HS algorithms and the MAB agent given as average running time (in seconds). The first column shows the different groups of instances. The second column tells the HS alternative providing the best performance. The letter indicates how the hitting vector is computed and the number indicates if it is augmented until it becomes maximal (1), or until it partially grows (2). In the initial implementation of HS, variation D was not created, hence why it does not show up in the column.

Table 1 shows the results of our experiments. Looking at column *Best alt.* we can see that there is no winning version among the different alternatives that we are considering. Looking at column *HS (best)* and *HS (worst)*, we can see that there is a significant difference between the best-performing alternative of the IHS compared to the worst. This means that if the user were to simply select an alternative from the 8, it could result in a very slow performance. Thus motivating the idea behind using a MAB bandit as a solution. Column *HS (average)* reports the average performance among all the tested HS alternatives. It can be seen that the average is often much higher than the best.

Column *MAB* shows the result of our MAB implementation. the first thing we can see is that in all of the cases, it performs better than the worst IHS for the given problem group, which means that the overhead of the MAB variables maintenance is not a problem. Next we can also see that it performs quite similarly to the average performance of the 8 algorithms for the given problem, and in some cases faster. This shows that we have a reliable agent which given any WCSP instance, can guarantee that it will solve it within an efficient time. Although the MAB is not able to outperform the best for none of the groups, in some of the cases where we have difficult instances, we can see that it performs twice as fast as the average.

7. Conclusions and Future Work

This paper presents the original idea of using a Multi-armed Bandit with the Implicit Hitting Set algorithm to apply it to Markov Random Fields and Bayesian networks. We present eight alternative implementations of the HS algorithm that can be used to construct the MAB. We see from the results that there is no winning strategy when it comes to only using the HS algorithm. Furthermore, when looking at the results we can conclude that the MAB agent performs better than the average over the eight HS alternatives, on different types of problems, including real ones. Through our work, we can ensure that given a WCSP instance, the agent can solve the problem efficiently.

We believe that with the results being positive and competitive with the HS approach there is quite some potential in using Reinforcement Learning for solving Graphical Models. To achieve better results, we would like to consider expanding the implementation to contain more arms. These arms would be based on other alternatives that the HS algorithm can contain. Such alternatives could provide the agent with additional possibilities of reaching the optimum. Based on the results, we can see that variation has been positive in ensuring the minimization of the overall time needed to solve the instance on average.

Another line of work can include shifting towards using Contextual bandits. From the results, we could see some patterns emerging on some types of instances performing better with a specific alternative of the HS algorithm. Although these might not always apply, providing the agent with more context about the instances instead of simply relying on the reward could lead to speeding up the time needed to solve it. This idea is motivated further by the fact that often an arm may not produce a core, which would result in gaining no reward, however, it has been crucial to reducing the gap between the bounds, putting the agent a step closer to solving the instance.

References

D. Allouche, S. de Givry, G. Katsirelos, T. Schiex, and M. Zytnicki. Anytime hybrid best-first search with tree decomposition for weighted CSP. In G. 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. URL https://doi.org/10.1007/978-3-319-23219-5_2.

C. Ansótegui, M. L. Bonet, and J. Levy. Scale-free random SAT instances. *Algorithms*, 15 (6):219, 2022. doi: 10.3390/A15060219. URL https://doi.org/10.3390/a15060219.

J. Berg, F. Bacchus, and A. Poole. Abstract cores in implicit hitting set maxsat solving. In L. Pulina and M. 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. URL https://doi.org/10.1007/978-3-030-51825-7_20.

A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In T. Balyo, N. Froleyks, M. Heule,

M. Iser, M. Järvisalo, and M. 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.

M. C. Cooper, S. de Givry, M. Sánchez-Fibla, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artif. Intell.*, 174(7-8):449–478, 2010. doi: 10.1016/j.artint.2010.02.001. URL https://doi.org/10.1016/j.artint.2010.02.001.

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

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

R. Dechter. *Reasoning with Probabilistic and Deterministic Graphical Models: Exact Algorithms, Second Edition.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019. doi: 10.2200/S00893ED2V01Y201901AIM041. URL https://doi.org/10.2200/S00893ED2V01Y201901AIM041.

E. Delisle and F. Bacchus. Solving weighted csps by successive relaxations. In C. 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. URL https://doi.org/10.1007/978-3-642-40627-0_23.

D. Koller and N. Friedman. *Probabilistic Graphical Models - Principles and Techniques.* MIT Press, 2009. ISBN 978-0-262-01319-2. URL http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11886.

J. Larrosa, C. Martinez, and E. 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.

J. Marques-Silva and I. Lynce. On improving MUS extraction algorithms. In K. A. Sakallah and L. 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. URL https://doi.org/10.1007/978-3-642-21581-0_14.

P. Saikko, C. Dodaro, M. Alviano, and M. Järvisalo. A hybrid approach to optimization in answer set programming. In M. Thielscher, F. Toni, and F. 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.

A. Slivkins. Introduction to multi-armed bandits. *Found. Trends Mach. Learn.*, 12(1-2):1–286, 2019. doi: 10.1561/2200000068. URL https://doi.org/10.1561/2200000068.

P. Smirnov, J. Berg, and M. Järvisalo. Pseudo-boolean optimization by implicit hitting sets. In L. 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. doi: 10.4230/LIPICS.CP.2021.51. URL https://doi.org/10.4230/LIPIcs.CP.2021.51.

P. Smirnov, J. Berg, and M. Järvisalo. Improvements to the implicit hitting set approach to pseudo-boolean optimization. In K. S. Meel and O. 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. doi: 10.4230/LIPICS.SAT.2022.13. URL https://doi.org/10.4230/LIPIcs.SAT.2022.13.

R. S. Sutton and A. G. Barto. *Reinforcement learning - an introduction.* Adaptive computation and machine learning. MIT Press, 1998. ISBN 978-0-262-19398-6. URL https://www.worldcat.org/oclc/37293240.