Constraints manuscript No.
(will be inserted by the editor)

Ultra-Weak Solutions and Consistency Enforcement in Minimax Weighted Constraint Satisfaction

Arnaud Lallouet · Jimmy H.M. Lee · Terrence W.K. Mak · Justin Yip

Abstract The task at hand is that of a soft constraint problem with adversarial conditions. By amalgamating the weighted and quantified constraint satisfaction frameworks, a Minimax Weighted Constraint Satisfaction Problem (formerly Quantified Weighted Constraint Satisfaction Problem) consists of a set of finite domain variables, a set of soft constraints, and a $\min$ or $\max$ quantifier associated with each of these variables. We formally define the framework, suggest three solution concepts, and propose a complete solver based on alpha-beta pruning techniques. We discuss in depth our novel definitions and implementations of node, arc and full directional arc consistency notions to help reduce search space on top of the basic tree search with alpha-beta pruning for solving ultra-weak solutions. In particular, these consistencies approximate the lower and upper bounds of the cost of a problem by exploiting the semantics of the quantifiers and reusing techniques from both Weighted and Quantified Constraint Satisfaction Problems. Lower bound computation employs standard estimation of costs in the sub-problems used in alpha-beta search. In estimating upper bounds, we propose two approaches based on the Duality Principle: duality of quantifiers and duality of constraints. The first duality amounts to changing quantifiers from $\min$ to $\max$, while the second duality re-uses the lower bound approximation functions on dual constraints to generate upper bounds. Experiments on three benchmarks

Arnaud Lallouet
Université de Caen, GREYC, Campus Côte de Nacre,
Boulevard du Maréchal Juin, BP 5186, 14032 Caen Cedex, France
E-mail: Arnaud.Lallouet@unicaen.fr

Jimmy H.M. Lee
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
E-mail: jlee@cse.cuhk.edu.hk

Terrence W.K. Mak
NICTA CRL, Tower A, 7 London Circuit, Canberra, ACT 2601, Australia
E-mail: Terrence.Mak@nicta.com.au

Justin Yip
Brown University, Box 1910, Providence, RI 02912, USA
E-mail: justin@cs.brown.edu

comparing basic alpha-beta pruning and the six consistencies from the two dualities are performed to confirm the feasibility and efficiency of our proposal.

Keywords constraint optimization, soft constraint satisfaction, minimax game search, consistency algorithms

1 Introduction

The task at hand is that of a constraint optimization problem with *adversaries* controlling parts of the variables. As an example, we begin with a generalized version of the Radio Link Frequency Assignment Problem (RLFAP) [8] consisting of assigning frequencies to a set of radio links located between pairs of sites, with the goal of preventing interferences. The problem has two types of constraints. One type prevents radio links that are close together from interfering with one another, by restricting the links not to take frequencies with absolute differences smaller than a threshold. In practice, the threshold is measured depending on the physical environment, and is often overestimated. The second type of constraints are technological constraints, where each constraint ensures the distance between frequencies of a radio link from site A to B and its reverse radio link from site B to A must be equal to a constant. If the problem is unsatisfiable, one approach is to find assignments violating the first type of constraints as little as possible. Suppose now a certain set of links are placed in unsecured areas, and *adversaries* (e.g. terrorists/spies) may hijack/control these links. One interesting question for this type of scenarios is to find frequency assignments such that we can minimize the degree of radio links affected for the worst possible case (i.e. finding the best-worst case). In practice, we may not even be able to immediately respond by re-adjusting the frequency assignments in order to minimize the interferences and planning how to defend is also important. The prime goal of our work is to understand how well we can defend against the worst adversaries for planning purposes.

The example is optimization in nature, and the adversaries originate from the uncontrollable frequencies being assigned on the links in unsecured areas. The question can be modeled as minimizing the interferences for all possible combinations of frequency adjustments the adversaries can control. One way to solve this problem is by tackling many COPs [2]/Weighted CSPs [17], where each of them minimizes the interferences conditioned on a specific combination of frequency adjustments controlled by the adversaries. Another way is to model the problem as a Quantified CSP [6] by finding whether there exists combinations of frequency adjustments for us for all frequency placements by the adversaries such that the costs of interferences is less than a cost k. To avoid solving multiple sub-problems, Minimax Weighted Constraint Satisfaction Problems (MWCSPs) (previously Quantified Weighted Constraint Satisfaction Problems) [21, 15] are proposed to tackle such problems, combining quantifier structures from Quantified CSPs to model the adversaries and soft constraints from Weighted CSPs to model costs information.

The generalized RLFAP described above can be viewed as a zero-sum two-player game played in two turns. When tackling such game problems, more specifically two-person zero-sum games with perfect information [33, 25], games can be solved

at different levels. Allis [1, 14] proposes three solving levels for games: *ultra-weakly solved*, *weakly solved*, and *strongly solved*. Ultra-weakly solved means the game-theoretic value of the initial position has been determined, which means we can determine the outcome of the scenario when both players are playing perfectly (i.e. best-worst case). Weakly solved means a strategy, noted as winning strategy [5] in Quantified CSPs, has been determined for the initial position to achieve the game-theoretic value against any opposition. Strongly solved is used for a game for which such a strategy has been determined for all legal positions. Once a game is solved at a stronger level, the game is automatically solved at weaker ones. Finding solutions at stronger levels, however, implies substantially higher computation requirements. In particular, in terms of space, ultra-weak solutions are linear in size, while the other two stronger ones are exponential. In bi-level programs, there are cases in which we can assume there is a unique optimum for the follower or we are concerned with only the moves for the leader [12]. Finding ultra-weak solutions for these cases are sufficient. In the generalized RLFAP example, we can see that operators setting the frequencies are classified as leaders and adversaries controlling the unsecured links are classified as followers. In adversarial game playing, many game search algorithms, e.g. minimax and alpha-beta [30], compute strategies assuming optimal plays to reduce computation costs. In fact, even determining just the ultra-weak solution in an offline manner is also an important and interesting line of research, e.g. a recent breakthrough on checkers [31].

We define three solution concepts: ultra-weak solutions, weak solutions, and strong solutions, corresponding to each of the three solving levels for Minimax Weighted CSPs, and our work focus on finding ultra-weak solutions. We describe how to adopt alpha-beta pruning to tackle the problem, and suggest two sufficient pruning conditions to achieve prunings and backtrackings. We also introduce novel consistency notions and algorithms for solving ultra-weak solutions, by approximating the lower and upper bounds of the cost of the problem. Lower bound computation employs standard estimation of costs in the sub-problems used in alpha-beta search. In estimating upper bounds, we adopt the Principle of Duality [24, 35] in (integer) linear programming, which suggests to convert an original (primal) problem to its dual form and tackle the problem using both forms. We consider two dualities: duality of quantifiers and duality of constraints. The first approach allows us to formulate upper bound approximation functions by changing quantifiers in the lower bound functions from $\min$ to $\max$, while the second approach re-uses the lower bound approximation functions on dual constraints to generate upper bounds. Algorithms and examples to explain these notions are given throughout the paper. Discussions on whether our proposed techniques are applicable to the computation of the two stronger solutions are also given. Experimental evaluations on three benchmarks are performed to compare six consistencies defined using the two dualities to confirm the feasibility and efficiency of our proposal.

This journal paper combines and improves two of our previous work: Lee, Mak, and Yip [21] and Lallouet, Lee, and Mak [15]. We have added and/or improved the following items:

1. Examples to illustrate the concepts of our consistencies,

2. Algorithms and pseudo-codes to enforce the proposed consistency notions,
3. Fuller and corrected proofs for our proposed lemmas and theorems,
4. Theoretical runtime results on our consistency algorithms, and
5. Extended experimental evaluations.

The rest of the paper is organized as follows. Section 2 gives the background definitions for Weighted CSPs and Quantified CSPs. We then define our framework Minimax Weighted CSPs in Section 3, followed by giving descriptions of the basic alpha-beta search. Based on the alpha-beta search, Section 4 proposes two sufficient pruning conditions to achieve prunings and backtrackings. Section 5 gives consistency notions and algorithms for solving Minimax Weighted CSPs, followed by performance evaluations on three benchmarks in Section 6. In Section 7, we conclude our work.

2 Background

We first give basic definitions for Weighted CSPs and Quantified CSPs.

A *Weighted Constraint Satisfaction Problem* [17] (WCSP) is a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$, where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a finite set of *variables* and $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of finite *domains* of possible values. We denote $x_i = v_i$ an *assignment* assigning value $v_i \in D_i$ to variable x_i, and the set of assignments $l = \{x_1 = v_1, x_2 = v_2, \ldots, x_n = v_n\}$ a *complete assignment* on variables in $\mathcal{X}$, where v_i is the value assigned to x_i. A *partial assignment* $l[S]$ is a projection of l onto variables in $S \subseteq \mathcal{X}$. $\mathcal{C}$ is a finite set of *(soft) constraints* (also called cost functions), each C_S of which represents a function mapping tuples corresponding to assignments on a subset of variables S, to a cost valuation structure $V(k) = ([0...k], \oplus, \leq)$. The structure $V(k)$ contains a set of integers $[0...k]$ with standard integer ordering $\leq$. Addition $\oplus$ is defined by $a \oplus b = \min(k, a + b)$. For any integer a and b where $a \geq b$, subtraction $\ominus$ is defined by $a \ominus b = a - b$ if $a \neq k$, and $a \ominus b = k$ if $a = k$. Note that for the rest of the paper, $+$ and $-$ refer to standard addition and subtraction while $\oplus$ and $\ominus$ refer to the addition and subtraction for the valuation structure. Without loss of generality, we assume the existence of $C_\varnothing$ denoting the lower bound of the minimum cost of the problem. If it is not defined, we assume $C_\varnothing = 0$. The *cost* of a complete assignment l in $\mathcal{X}$ is defined as:

$$cost(l) = C_\varnothing \oplus \bigoplus_{C_s \in \mathcal{C}} C_s(l[S])$$

A complete assignment l on $\mathcal{X}$ is *feasible* if $cost(l) < k$, and is a *solution* of a WCSP if l has the minimum cost among all tuples.

A *Quantified Constraint Satisfaction Problem* [6] (QCSP) $\mathcal{P}$ is a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q})$, where $\mathcal{X} = (x_1, \ldots, x_n)$ is an ordered finite sequence of variables, $\mathcal{D} = (D_1, \ldots, D_n)$ is an ordered sequence of finite domains, $\mathcal{C} = \{C_1, \ldots, C_e\}$ is a finite set of constraints, and $\mathcal{Q} = (Q_1, \ldots, Q_n)$ is a quantifier sequence in which each Q_i is either $\exists$ (existential, 'there exists') or $\forall$ (universal, 'for all') associated with x_i. A *constraint* $C_k \in \mathcal{C}$ consists of a sequence $\mathcal{X}_k = (x_{k_1}, \ldots, x_{k_r})$ of $r > 0$ variables

s.t. $\mathcal{X}_k$ is a subsequence of $\mathcal{X}$. C_k has an associated set $A[C_k] \subseteq D_{k_1} \times \ldots \times D_{k_r}$ of *tuples* which specify allowed combinations of values for the variables in $\mathcal{X}_k$. Let $\mathrm{firstx}(\mathcal{P})$ be a function returning the first unassigned variable in the variable sequence. If there are no such variables, it returns $\perp$. The *semantics* of a QCSP $\mathcal{P}$ is defined recursively as follows:

(1) In case $\mathrm{firstx}(\mathcal{P}) = \perp$, if all constraints $C_k \in \mathcal{C}$ are satisfied, $\mathcal{P}$ is *satisfiable*; and if any constraint fails, $\mathcal{P}$ is *unsatisfiable*.
(2) Otherwise, let $\mathrm{firstx}(\mathcal{P}) = x_i$. If $Q_i = \exists$ then $\mathcal{P}$ is *satisfiable* iff there exists a value $a \in D_i$ such that the simplified problem $\mathcal{P}$ with a assigned to x_i is satisfiable. If $Q_i = \forall$ then $\mathcal{P}$ is *satisfiable* iff for all values $a \in D_i$ the simplified problem $\mathcal{P}$ with a assigned to x_i is satisfiable.

3 Minimax Weighted Constraint Satisfaction Problems

In this section, we give definitions and semantics of MWCSPs. We then further describe the alpha-beta search for MWCSPs.

3.1 Definitions and Semantics

Standard Weighted CSPs are minimization in nature. We aim at optimizing problems with adversarial conditions by modeling adversaries using $\max$ quantifiers. A *Minimax Weighted Constraint Satisfaction Problem* (MWCSP) [21, 15] $\mathcal{P}$ is a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q}, k)$, where $\mathcal{X} = (x_1, \ldots, x_n)$ is defined as an ordered sequence of *variables*, $\mathcal{D} = (D_1, \ldots, D_n)$ is an ordered sequence of finite *domains*, $\mathcal{C}$ is a set of *soft constraints*, $\mathcal{Q} = (Q_1, \ldots, Q_n)$ is a *quantifier sequence* where Q_i is either $\max$ or $\min$ associated with x_i, and k is the global upper bound. We re-use the definition of assignments, partial and complete assignments, (soft) constraints, and costs of a complete assignment for Weighted CSPs.

In an MWCSP, ordering of variables is important. Without loss of generality, we assume variables are ordered by their indices. We define a variable with $\min$ ($\max$ resp.) quantifier to be a minimization variable (maximization variable resp.). Let $\mathcal{P}[x_{i_1} = a_{i_1}][x_{i_2} = a_{i_2}] \ldots [x_{i_m} = a_{i_m}]$ be the *sub-problem* obtained from $\mathcal{P}$ by assigning value a_{i_1} to variable x_{i_i}, assigning value a_{i_2} to variable x_{i_2},..., assigning value a_{i_m} to variable x_{i_m}. We re-use the function firstx from the definition of Quantified CSPs. The *aggregated costs* of an MWCSP $\mathcal{P}$, *A-cost($\mathcal{P}$)*, is defined recursively as follows:

$$\text{A-cost}(\mathcal{P}) = \begin{cases} cost(l), \text{ if } \mathrm{firstx}(\mathcal{P}) = \perp \\ \max(\mathbb{M}_i), \text{ if } \mathrm{firstx}(\mathcal{P}) = x_i \text{ and } Q_i = \max \\ \min(\mathbb{M}_i), \text{ if } \mathrm{firstx}(\mathcal{P}) = x_i \text{ and } Q_i = \min \end{cases}$$

where l is the complete assignment of the completely assigned problem $\mathcal{P}$ (i.e. $\mathrm{firstx}(\mathcal{P}) = \perp$), and $\mathbb{M}_i = \{\text{A-cost}(\mathcal{P}[x_i = v]) | v \in D_i\}$. An MWCSP $\mathcal{P}$ is *satisfiable* iff $\text{A-cost}(\mathcal{P}) < k$. We define a *block* of variables in an MWCSP $\mathcal{P}$ to be

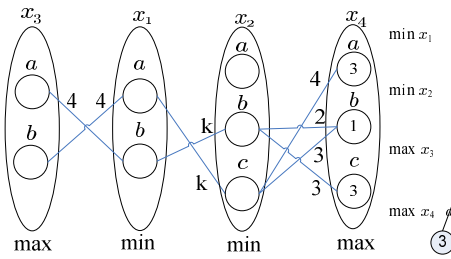

Fig. 1 Constraints for Example 1

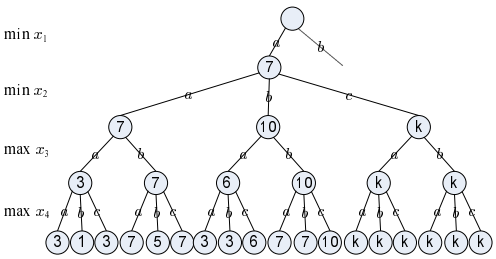

Fig. 2 Labeling Tree for Example 1

a maximal subsequence of variables in $\mathcal{X}$ which has the same quantifiers. Changing the variable ordering within the same block of variables does not change the A-cost of an MWCSP.

We now define three solution concepts [15] for MWCSPs based on the definition of A-costs. An *ultra-weak solution* of an MWCSP $\mathcal{P}$ is a complete assignment $\{x_1 = v_1, \ldots, x_n = v_n\}$ s.t. A-cost$(\mathcal{P})$ = A-cost$(\mathcal{P}[x_1 = v_1] \ldots [x_i = v_i]), \forall 1 \leq i \leq n$. Finding an ultra-weak solution corresponds to finding one scenario when both players are playing perfectly. To capture weak (strong resp.) solutions, we re-use the concept of winning strategies [5]. Without loss of generality, we assume the max player is the adversary. A *weak solution* (*strong solution* resp.) is a set of functions $\mathcal{F}$, where each function $f_i \in \mathcal{F}$ corresponds to a min variable x_i. Let G_i be the set of domains of max variables (all variables resp.) preceding x_i, i.e. $G_i = \{D_j \in \mathcal{D} | Q_j = \max \wedge j < i\}$ ($G_i = \{D_j \in \mathcal{D} | j < i\}$ resp.). We define $f_i : \times_{D_j \in G_i} D_j \mapsto D_i$. If G_i is an empty set, then f_i is a constant function returning values from D_i. Let $\mathcal{P}'$ be a sub-problem of an MWCSP $\mathcal{P}$, where the next unassigned variable x_i is a min variable, and l be the set of assigned values for max variables (all variables resp.) x_j where $j < i$. For weak solutions, we can further assume the assigned values of min variables x_k where $k < i$ in $\mathcal{P}'$ follow f_k. We require all f_i to satisfy: A-cost$(\mathcal{P}'[x_i = f_i(l)])$ = A-cost$(\mathcal{P}')$. In other words, we require $f_i(l)$ to return the best value for the min player, and the set of functions $\mathcal{F}$ will then be a best strategy for the min player. *This paper focuses on tackling ultra-weak solutions.*

Example 1 We use the generalized Radio Link Frequency Assignment Problem introduced in the previous section as an example to illustrate our concepts throughout the paper. The problem consists of four links l_1, l_2, l_3, and l_4. Two of the links l_1 and l_2 connect sites A and B, and the other two links l_3 and l_4 connect sites B and C. Link l_2 (l_4 resp.) is the reverse link for l_1 (l_3 resp.). There is a variable x_i in the MWCSP $\mathcal{P}$ for each link l_i, which is used to represent the chosen frequency for link l_i. Site C is not secure and links l_3 and l_4 are subject to control. We need to pay costs if two links interfere with each other. Therefore, we want to find frequency assignments for l_1 and l_2 such that we can minimize the total costs for interference in the worst case. We set the quantifier sequence in $\mathcal{P}$ as $(Q_1 = \min, Q_2 = \min, Q_3 = \max, Q_4 = \max)$. For simplicity, we assume links l_1 and l_3 have two frequency choices, and the other two links have three. We measure the costs for interference only for links l_1 and l_3, and links l_2 and l_4. These costs will be modeled by constraints on variables x_1 and x_3, and also on variables x_2 and x_4. In addition, we maintain the technological constraint

between links l_1 and l_2, which will be modeled by a binary constraint on variables x_1 and x_2. We can also add unary constraints to the problem to indicate known preferences on the link frequencies. In this example, we add one such constraint to x_4. Note that the example is simplified for illustrative purposes. Figure 1 indicates there is one unary constraint C_4 and three binary constraints $C_{1,2}, C_{1,3}$, and $C_{2,4}$. For the unary constraint, non-zero unary costs are depicted inside a circle and domain values are placed above the circle. For binary constraints, non-zero binary costs are depicted as labels on edges connecting the corresponding pair of values. Only non-zero costs are shown. We set the global upper bound k to be 11. By following the partial labeling tree in Figure 2, we can easily infer the $\mathrm{A\text{-}cost}$ of the subproblem $\mathcal{P}' = \mathcal{P}[x_1 = a]$ is 7, and $\{x_1 = a, x_2 = a, x_3 = b, x_4 = a\}$ is one of the ultra-weak solutions for the sub-problem $\mathcal{P}'$.

From the problem definitions of Minimax Weighted CSPs, we can observe that both Weighted CSPs and Quantified CSPs are special cases of Minimax Weighted CSPs.

Lemma 1 *A Weighted CSP [17] $\mathcal{P}$ can be transformed by polynomial-time (Karp) reduction [3] to a Minimax Weighted CSP $\mathcal{P}'$. Finding the* $\mathrm{A\text{-}cost}$ *of $\mathcal{P}'$ is equivalent to finding the minimum cost of $\mathcal{P}$ [21].*

By constructing a Minimax Weighted CSP $\mathcal{P}'$ with the same set of variables, domains, and soft constraints from the Weighted CSP $\mathcal{P}$ and setting all quantifiers of $\mathcal{P}'$ to $\min$ quantifiers, it is easy to observe that finding the $\mathrm{A\text{-}cost}$ of $\mathcal{P}'$ essentially finds the minimum cost of $\mathcal{P}$.

Lemma 2 *A Quantified CSP [6] $\mathcal{P}$ can be transformed by polynomial-time (Karp) reduction [3] to a Minimax Weighted CSP $\mathcal{P}'$. Finding the* $\mathrm{A\text{-}cost}$ *of $\mathcal{P}'$ is equivalent to determining the satisfiability of $\mathcal{P}$ [21].*

We construct a Minimax Weighted CSP $\mathcal{P}'$ which holds the same set of variables and domains as in the Quantified CSP $\mathcal{P}$. For the quantifiers, if a variable in $\mathcal{P}$ has an $\exists$ quantifier (a $\forall$ quantifier resp.) , the same variable in $\mathcal{P}'$ will have a $\min$ ($\max$ resp.) quantifier. The final step in the transformation involves transforming constraints in $\mathcal{P}$ to soft constraints in $\mathcal{P}'$. For every constraint C in $\mathcal{P}$, we construct a soft constraint C' for $\mathcal{P}'$ on the same set of variables, where C' returns a cost of 0 (a cost of k resp.) on the same set of assignments if C is satisfiable (unsatisfiable resp.). We set k to any positive integer larger than 0. It is not hard to check if the $\mathrm{A\text{-}cost}$ of $\mathcal{P}'$ is 0, then $\mathcal{P}$ is satisfiable; otherwise, $\mathcal{P}'$ is unsatisfiable.

Corollary 1 *Finding the A-costs and ultra-weak solutions of Minimax Weighted CSPs are PSPACE-hard [21].*

The corollary follows from the fact that Quantified CSPs are PSPACE-complete [6]. Computing ultra-weak solutions for Minimax Weighted CSPs essentially computes the $\mathrm{A\text{-}costs}$, and finding the $\mathrm{A\text{-}costs}$ of Minimax Weighted CSPs are PSPACE-hard (by Lemma 2). Therefore, finding ultra-weak solutions are also PSPACE-hard. A special case is that if all the quantifiers of an MWCSP are $\min$ quantifiers, finding an

ultra-weak solution is equivalent to finding a complete assignment l with the minimum costs (i.e. $\mathrm{argmin}_l\, cost(l)$). The problem reduces to a Weighted CSP, which is NP-hard.

It is worth to note that Minimax Weighted CSPs are also sub-classes of the multi-operator framework [26] and can be classified also as a kind of sequential decision making problems [27]. Since our work directly defined based on Weighted CSPs and Quantified CSPs, we naturally focus on combing and re-use ideas and techniques from these two frameworks.

3.2 Alpha-beta Prunings in Branch & Bound

Minimax Weighted CSPs can be solved by applying alpha-beta pruning in branch and bound search [21, 15], by treating $\max$ and $\min$ variables as $\max$ and $\min$ players respectively. Note that alpha-beta pruning is not new and has long been introduced as pruning strategies in the game solving community. Admissible heuristics and/or monotonic function have also been proposed to incorporate with alpha-beta prunings [32, 28]. Our work can be seen as re-applying these classical principles and ideas from the game solving community to MWCSPs, which is a constraint-based framework. One main theme of our work is to show how these ideas could be combined with state-of-the-art constraint propagation techniques from both Weighted CSPs and Quantified CSPs.

Alpha-beta pruning utilizes two bounds, α and β, for storing the current best costs for $\max$ and $\min$ players. We rename α and β as lower lb and upper ub bounds to fit with the common notations for bounds in constraint and integer programming. Figure 3 shows the alpha-beta search for our Minimax Weighted CSP solver. Function `local_consistency` at line 4 (enclosed in the grey box) is used to invoke routines for enforcing local consistencies. Since we will introduce local consistencies and their enforcing algorithms in later sections, we now skip the explanation for this function. At current stage, we assume the function `local_consistency` is empty and only returns a constant return value `NO_BTK`. In this case, we can further ignore line 5 since the if-condition will always avoid executing the return statement.

The alpha-beta algorithm starts by initializing lb (ub resp.) to -1 (k resp.), and these two bounds will be maintained during assignments. Therefore, the search starts with `alpha_beta`$(\mathcal{P}, -1, k)$. Line 2 is the base case in which all variables are bounded. In this case, alpha-beta pruning will invoke the routine `cost`, which returns the cost of the current complete assignment. Lines 6 to 12 give the main routine of the traditional alpha-beta pruning algorithm. We only explain the cost for the $\min$ quantifier, since that of $\max$ is similar. The for loop evaluates all sub-problems $\mathcal{P}[x_i = v]$ by recursively invoking the alpha-beta algorithm. Since the goal is to find a minimum value, the upper bound is updated. When the upper bound is less than the lower bound (line 11), it triggers the short-cut to break out of the remaining search since every value returned by subsequent calls will be dominated by the current bounds. The function `alpha_beta` ends by returning the upper bound for the $\min$ quantifier (line 12).

```
1  function alpha_beta(P,lb,ub):
2    if firstx(P) == ⊥: return cost(P)
3    i = firstx(P)
4    state = local_consistency()*
5    if state != NO_BTK: return (state == UB_BTK)?ub:lb
6    for v in Di:
7      if Qi == min:
8        ub = min(ub, alpha_beta(P[Xi=v],lb,ub))
9      else:
10       lb = max(lb, alpha_beta(P[Xi=v],lb,ub))
11     if ub <= lb: break
12   return (Qi == min)?ub:lb
```

Fig. 3 Alpha-beta for Minimax Weighted CSPs

Note that when line 4 and 5 are skipped, the algorithm is essentially an ordinary alpha-beta pruning algorithm. Hence, we could argue the soundness of the algorithm by re-using the argument for alpha-beta prunings. Recall lb and ub are used to store the current best costs for $\max$ and $\min$ players. Suppose the alpha-beta algorithm is now exploring a problem $\mathcal{P}$ with upper bound ub. We assume the first quantifier to be $\min$. The algorithm will only skip evaluating sub-problems $\mathcal{P}[x_i = v]$ when the condition at line 11 is triggered. For line 11 to be triggered, there must exists another sub-problem $\mathcal{P}[x_i = u], u \neq v$ which gives a better upper bound ub' comparing to the original upper bound ub, and ub' is lower than or equal to lb. According to two-player zero-sum game, this essentially means that the min player now has found a strategy by playing $x_i = u$ to achieve a cost lower than or equal to lb, which is the current best found costs for the max player. Therefore, the max player has no hope on finding a cost larger than lb by following the current branch regardless on the costs of the alpha-beta search on sub-problems $\mathcal{P}[x_i = v]$. In other words, the max player will tend to follow a previous searched branch leading to his/her current best cost lb and alpha-beta will therefore prune all the sub-problems $\mathcal{P}[x_i = v]$. Similar reasonings can be applied when the first quantifier is $\max$.

Note that by utilizing the alpha-beta search in Figure 3, we cannot find all ultra-weak solutions. Assume we have already found the first ultra-weak solution. Suppose now the alpha-beta search encounters the second ultra-weak solution. It is not hard to observe that Line 11 will immediately causing the solver to backtrack as the costs of the second solution are equal to the costs of the found solution. If we want to gather all ultra-weak solutions, we may need to relax the backtracking condition by modify line 11 to change the inequality to a strict inequality (i.e. from $\leq$ to $<$).

4 Pruning and Backtracking Conditions

In traditional CSPs, we enforce different levels of consistency to prune infeasible domain values and hence reduce the search space. In Weighted CSPs, consistency algorithms further take the costs of constraints into account. Various consistency notions (e.g. NC*, AC* [17], FDAC*, EDAC*, OSAC, and VAC [9]) have been proposed

and proven to be useful in improving solver performance. Such techniques, however, cannot be directly applied to Minimax Weighted CSPs since the quantifiers change the semantics of constraints. In particular, applying these consistency algorithms on constraints constraining on $\max$ variables may result in unsound prunings. We need to devise consistency notions for Minimax Weighted CSPs that take quantifiers into account. We first study conditions on when a pruning/backtracking is sound.

To prune values of Minimax Weighted CSPs, the main idea is that if the A-cost of a sub-problem is greater than or equal to the upper bound ub (less than or equal to the lower bound lb resp.), we can apply pruning techniques based on alpha-beta search. Let $\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]$ denote the subproblem $\mathcal{P}[x_1 = v_1][x_2 = v_2]...[x_{i-1} = v_{i-1}][x_i = v]$. Formally, we consider two conditions: $\exists v \in D_i \;\; s.t. \;\; \forall v_1 \in D_1, ..., v_{i-1} \in D_{i-1}$:

$$\text{A-cost}(\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]) \geq ub \tag{1}$$

$$\text{A-cost}(\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]) \leq lb \tag{2}$$

where ub and lb are the upper and lower bounds in alpha-beta pruning respectively. When either of the above conditions is satisfied, we can apply prunings/backtrackings according to Table 1 [21, 15].

Table 1 When can we prune/backtrack

A-cost	$\geq ub$	$\leq lb$
$Q_i = \min$	prune v	backtrack
$Q_i = \max$	backtrack	prune v

Theorem 1 *Suppose we were given a Minimax Weighted CSP $\mathcal{P}$. If Condition (1)/(2) for $\mathcal{P}$ is satisfied, applying prunings and backtrackings in alpha-beta pruning according to Table 1 is sound. [21]*

Proof (Sketch) Reasons to perform prunings and backtracking for $\min$ and $\max$ are symmetrical. We only describe the case when $Q_i = \min$. Suppose Condition (1) holds. We consider A-cost(s) for sub-problems $\mathcal{P}[x_{1..i-1} = v_{1..i-1}]$. Without loss of generality, we write $\mathcal{P}_{i-1}$ to represent one of these sub-problems $\mathcal{P}[x_{1..i-1} = v_{1..i-1}]$ by fixing values $v_1 \in D_1$, $v_2 \in D_2$, ..., $v_{i-1} \in D_{i-1}$. We will see that the proof using $\mathcal{P}_{i-1}$ applies for all sub-problems $\mathcal{P}[x_{1..i-1} = v_{1..i-1}]$, regardless of which values we fix. Given $Q_i = \min$, we obtain:

$$\text{A-cost}(\mathcal{P}_{i-1}) = \min_{a \in D_i} \mathcal{P}_{i-1}[x_i = a]$$

If $\text{A-cost}(\mathcal{P}_{i-1}) < ub$, the following must be true:

$$\exists v' \in D_i \text{ where } v' \neq v \;\; s.t. \;\; \text{A-cost}(\mathcal{P}_{i-1}[x_i = v']) < ub$$

Pruning value v does not change the A-cost of P_{i-1}. If $\text{A-cost}(\mathcal{P}_{i-1}) \geq ub$, i.e. $\mathcal{P}_{i-1}$ must not lead to any ultra-weak solutions, the following must be true:

$$\forall v' \in D_i, \text{A-cost}(\mathcal{P}_{i-1}[x_i = v']) \geq ub$$

After pruning value v, either domain wipe out occurs or A-cost$(\mathcal{P}_{i-1}) \geq ub$. For both cases, the sub-problem $\mathcal{P}_{i-1}$ cannot lead to solutions. Combining the two cases, pruning value v does not change the problem from unsatisfiable to satisfiable (and vice versa).

We now discuss Condition (2). Similar to the previous case, we consider the A-cost for these sub-problems $\mathcal{P}[x_{1..i-1} = v_{1..i-1}]$, and we fix $\mathcal{P}_{i-1}$ to be one of these sub-problems similarly. Given $Q_i = \min$, we obtain:

$$\text{A-cost}(\mathcal{P}_{i-1}) = \min_{a \in D_i} \mathcal{P}_{i-1}[x_i = a]$$

By Condition (2), $\mathcal{P}_{i-1}[x_i = v] \leq lb$ holds, and therefore:

$$\text{A-cost}(\mathcal{P}_{i-1}) \leq lb$$

Recall A-cost$(\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v]) \leq lb$ applies regardless on which value $v_1, v_2, \ldots, v_{i-1}$ we fix. Therefore, we obtain:

$$\forall v_1 \in D_1, ..., v_{i-1} \in D_{i-1}, \text{A-cost}(\mathcal{P}[x_{1..i-1} = v_{1..i-1}]) \leq lb$$

We can easily obtain the following result using the definition of A-cost: A-cost$(\mathcal{P}) \leq lb$. Therefore, we can perform backtrack as the current search must lead to ultra-weak solutions in which the $\max$ player does not have a better move. □

Example 2 Suppose we were given a Minimax Weighted CSP $\mathcal{P}$ with the ordered set of variables $\{x_1,x_2,x_3\}$, domains $D_1 = \{a, b, c\}, D_2 = \{a, b\}$, and $D_3 = \{a, b, c\}$, the ordered set of quantifiers $\{Q_1 = \max, Q_2 = \min, Q_3 = \max\}$, and the global upper bound 10. Suppose the A-cost for sub-problem $\mathcal{P}[x_1 = a]$ is 1. Figure 4 shows the upper bound ub, lower bound lb, and the A-costs for the remaining sub-problems. By inspecting the figure, we can see that Condition (2) holds:

$$\exists a \in D_3, \forall v_1 \in D_1, \forall v_2 \in D_2, \text{A-cost}(\mathcal{P}[x_1 = v_1][x_2 = v_2][x_3 = a]) \leq lb$$

By Table 1, we can prune value a of x_3. We can observe that ultra-weak solutions must not contain the assignment $x_3 = a$, and therefore, we can prune the value. After pruning value a of x_3, we also observe that Condition (1) holds:

$$\exists b \in D_2, \forall v_1 \in D_1, \text{A-cost}(\mathcal{P}[x_1 = v_1][x_2 = b]) \geq ub$$

Similarly, we can prune value b of x_2.

Example 3 Suppose the ordered set of quantifiers of Example 2 is replaced by $\{Q_1 = \max, Q_2 = \max, Q_3 = \min\}$, and the A-cost for sub-problem $\mathcal{P}[x_1 = a]$ remains unchanged (A-cost$(\mathcal{P}[x_1 = a]) = 1$). Figure 5 shows the upper bound ub, lower bound lb, and the A-costs for the remaining sub-problems. Costs for each complete assignment remain the same as in Example 2. The only difference is the modified A-costs for sub-problems, resulting from the change in quantifiers. By inspecting the figure, we can observe that Condition (2) holds: $\exists a \in D_3 \;\; s.t. \;\; \forall v_1 \in D_1, \forall v_2 \in D_2,$

$$\text{A-cost}(\mathcal{P}[x_1 = v_1][x_2 = v_2][x_3 = a]) \leq lb$$

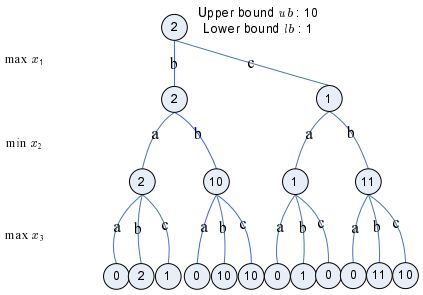

Fig. 4 Labeling Tree for Example 2

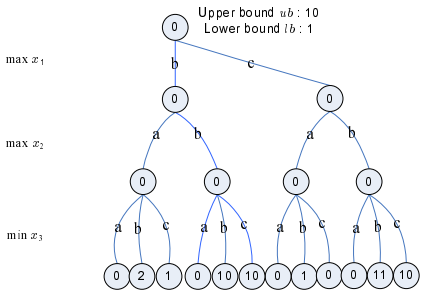

Fig. 5 Labeling Tree for Example 3

As $Q_3 = \min$, all the A-costs for sub-problems $\mathcal{P}[x_1 = v_1][x_2 = v_2], \forall v_1 \in D_1, v_2 \in D_2$ must be less than or equal to the lb. By induction, we can conclude sub-problems $\mathcal{P}[x_1 = v_1], \forall v_1 \in D_1$ must be less than or equal to the lb, and obtain $\text{A-cost}(\mathcal{P}) \leq lb$. Therefore following Table 1, the solver can backtrack.

One way to check Condition (1)/(2) is to find the exact value of the A-cost for each sub-problem, which is computationally expensive. The problem is essentially equivalent to determining if a variable assignment is a solution of a classical CSPs in general, which is NP-hard. A common technique in constraint programming is to formulate consistency notions and devise efficient algorithms, which aim at extracting and making useful information in a problem explicit (e.g. pruning and cost information).

In Minimax Weighted CSPs, we aim at extracting cost information in the form of *bounds* helping us to check whether Condition (1)/(2) is satisfied. Checking these conditions helps us to backtrack or identify non-solution values from domains early in the search. To extract these bounds, we introduce two approximating functions. Function $\text{ubaf}(\mathcal{P}, x_i = v)$ ($\text{lbaf}(\mathcal{P}, x_i = v)$ resp.) [21,15] is an upper bound (a lower bound resp.) approximation function if it approximates the A-cost for the set S of sub-problems, where $S = \{\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v] | v_1 \in D_1, \ldots, v_{i-1} \in D_{i-1}\}$ s.t.:

$$\forall \mathcal{P}' \in S, \text{A-cost}(\mathcal{P}') \leq \text{ubaf}(\mathcal{P}, x_i = v)$$
$$(\forall \mathcal{P}' \in S, \text{A-cost}(\mathcal{P}') \geq \text{lbaf}(\mathcal{P}, x_i = v) \text{ resp.})$$

Function $\text{ubaf}(\mathcal{P}, x_i = v)$ is *tight* iff $\max_{\mathcal{P}' \in S} \text{A-cost}(\mathcal{P}') = \text{ubaf}(\mathcal{P}, x_i = v)$. Similarly, function $\text{lbaf}(\mathcal{P}, x_i = v)$ is *tight* iff $\min_{\mathcal{P}' \in S} \text{A-cost}(\mathcal{P}') = \text{lbaf}(\mathcal{P}, x_i = v)$.

Corollary 2 *Suppose we were given a Minimax Weighted CSP $\mathcal{P}$. If $\text{ubaf}(\mathcal{P}, x_i = v) \leq lb$, we can prune value v of variable x_i if $Q_i = \max$, and perform backtrack if $Q_i = \min$ in alpha-beta pruning. If $\text{lbaf}(\mathcal{P}, x_i = v) \geq ub$, we can prune value v of variable x_i if $Q_i = \min$, and perform backtrack if $Q_i = \max$ in alpha-beta pruning. [21, 15]*

Proof (Sketch) We can check that the following holds from the definition of approximating functions.

Condition (1): $\mathrm{lbaf}(\mathcal{P}, x_i = v) \geq ub \implies$
$\forall v_1 \in D_1, v_2 \in D_2, \ldots, v_{i-1} \in D_{i-1}, \mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v] \geq ub$

Condition (2): $\mathrm{ubaf}(\mathcal{P}, x_i = v) \leq lb \implies$
$\forall v_1 \in D_1, v_2 \in D_2, \ldots, v_{i-1} \in D_{i-1}, \mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v] \leq lb$

We can then apply Table 1 according to the corresponding conditions. □

Note that the two functions we define: $\mathrm{lbaf}()$ and $\mathrm{ubaf}()$ are a specific kind of admissible heuristic functions [32] used by the game-solving community in alpha-beta search. However, the primary goal in defining such functions is to assist us in devising constraint propagation algorithms (to prune values and/or backtrack) since our framework is a constraint based framework. Exploring other general game solving strategies/principles which could utilize these two functions will be left as future work.

The main idea is that if we can implement $\mathrm{lbaf}()/\mathrm{ubaf}()$ with good and efficient approximations, we can identify non-solution values from variable domains or perform backtracking earlier in search. We show the high-level propagation routine for function `local_consistency` in our solver in Figure 6 to achieve prunings/backtracking according to Table 1 by utilizing $\mathrm{lbaf}()$ and $\mathrm{ubaf}()$. Line 4 to 17 show the main propagation loop, which continues to propagate until no more values can be pruned. Since finding the exact A-cost is difficult, the algorithm utilize the estimated bounds by calling the two approximation functions ($\mathrm{lbaf}()$ in line 8 and $\mathrm{ubaf}()$ in line 13). After computing the two estimated bounds: `ap_lb` for the estimated lower bound and `ap_ub` for the estimated upper bound, we then perform prunings/backtrackings according to Table 1. Recall in Table 1, there are a total of four different cases to handle. Line 10, 12, 15, and 16 show the implementation of the top left, bottom left, top right, and bottom right cases in Table 1 respectively. To perform prunings, we use the function `P[xj != u]` to prune a value u from domain D_j of variable x_j. To perform backtrackings, we use two return values: `UB_BTK` and `LB_BTK` to distinguish the two different backtracks (i.e. the bottom left and top right case) in Table 1 . If there are no backtrackings, the function returns `NO_BTK`.

Function `local_consistency()` will be called at line 4 in the main alpha-beta search (in Figure 3) before entering the main routine. If no early backtrackings could be found and detected by checking the conditions (Condition (1) / (2)) in `local_consistency()`, the main alpha-beta search will then enters the main searching routine. If early backtracking is found (top right or bottom left case in Table 1), we distinguish the two cases by returning different flags. For the top-right (bottom-left resp.) case, the function will return `LB_BTK` (`UB_BTK` resp.). This will allow the alpha-beta algorithm backtrack by returning lb (ub resp.) to indicate that the min player (max player resp.) has a move in the future which is able to guarantee the costs lower than (higher than resp.) then the current best cost of the max player (min player resp.). By returning lb (ub resp.), we can force the alpha-beta search to backtrack to the latest assigned $\max$ variable ($\min$ variable resp.) to allow the $\max$ player ($\min$ player resp.) finding a better action.

```
function local_consistency():
  i = firstx(P)
  changed = true
  while changed:
    changed = false
    for j in i..n:
      for u in Dj:
        ap_lb = lbaf(P, xj = u)
        if ub <= ap_lb:
          if Qj == min: P = P[xj != u]
                        changed = true
          if Qj == max: return UB_BTK
        ap_ub = ubaf(P, xj = u)
        if ap_ub <= lb:
          if Qj == min: return LB_BTK
          if Qj == max: P = P[xj != u]
                        changed = true
  return NO_BTK
```

Fig. 6 The high-level propagation routine using the two approximation functions

5 Consistency Techniques

This section discusses how we utilize costs information from unary constraints and binary constraints to formulate node and (full directional) arc consistencies. We start by giving an $lbaf()$ for node consistency called $nc_{lb}()$, which formulates lower bounds by gathering unary costs. We then further describe a stronger $lbaf()$ for (full directional) arc consistency called $ac_{lb}()$. To approximate upper bounds, we propose two approaches by utilizing the Duality Principle: duality of quantifiers and duality of constraints. In the last part, we discuss how to strengthen our consistency notions, by incorporating techniques from Weighted CSPs.

We write C_i for the unary constraint on variable x_i, $C_{i,j}$ for the binary constraint on variables x_i and x_j where $i < j$, $C_i(u)$ for the cost returned by the unary constraint when u is assigned to x_i, and $C_{i,j}(u, v)$ for the cost returned by the binary constraint when u and v are assigned to x_i and x_j respectively. To simplify our notations, we write the minimum costs $\min_{u \in D_j} C_j(u)$ and maximum costs $\max_{u \in D_j} C_j(u)$ of a unary constraint C_j as $\min C_j$ and $\max C_j$ respectively. We further write $Q_j C_j$ to mean $\min C_j$ if $Q_j = \min$, and $\max C_j$ if $Q_j = \max$. Algorithms for finding $\min C_j$, $\max C_j$, and $Q_j C_j$ (via functions `min()`, `max()`, and `Q()` resp.) are shown in Figure 7. The time complexity of these algorithms are in O(d), where d is the maximum variable domain size.

5.1 Node Consistency: Lower Bound

We first give the definition for $nc_{lb}()$. We then sketch the proof showing $nc_{lb}()$ is an $\mathrm{lbaf}()$ using two lemmas. Without loss of generality, we now consider unary MWCSPs, which are MWCSPs with *unary constraints only*. We will show that computing A-costs for any sub-problems of unary MWCSPs are efficient (linear time), and

```
function min(Ci):
  minimumCost = +∞
  for u in Di:
    if Ci(u) < minimumCost:  minimumCost = Ci(u)
  return minimumCost
function max(Ci):
  maximumCost = -∞
  for u in Di:
    if Ci(u) > maximumCost:  maximumCost = Ci(u)
  return maximumCost
function Q(Ci):
  if Qi == min: return min(Ci)
  if Qi == max: return max(Ci)
```

Fig. 7 Common routines for finding minimum and maximum costs for unary constraints

therefore, computing the lower bound for these sub-problems are efficient. We then show using the same procedure on general MWCSPs, by viewing unary constraints only, the bound is still correct.

Definition 1 The $nc_{lb}(\mathcal{P}, x_i = v)$ function approximates the A-cost for a set S of sub-problems $\{\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v] | v_1 \in D_1, \ldots, v_{i-1} \in D_{i-1}\}$. Define

$$nc_{lb}(\mathcal{P}, x_i = v) \equiv C_\varnothing \oplus (\bigoplus_{j:j<i} \min C_j) \oplus (C_i(v)) \oplus (\bigoplus_{j:i<j} Q_j C_j)$$

where $Q_j \in \mathcal{Q}$ is the quantifier for variable x_j where $j > i$.

Lemma 3 *The* A-cost *of an MWCSP $\mathcal{P}$ with only unary constraints is equal to $\bigoplus_{i=1}^{n} Q_i C_i$ [15].*

The proof of Lemma 3 follows directly from the definition of A-costs for MWCSPs.

Lemma 4 *Suppose we were given an MWCSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q}, k)$. Let E to be an arbitrary subset of constraints from $\mathcal{C}$ and we define $\mathcal{P}'$ to be an MWCSP obtained from $\mathcal{P}$ by removing all constraints in E (i.e. $\mathcal{P}' = (\mathcal{X}, \mathcal{D}, \mathcal{C} - E, \mathcal{Q}, k)$).*

$$\text{A-cost}(\mathcal{P}') \leq \text{A-cost}(\mathcal{P})$$

Proof (Lemma 4) We first consider the simplified case where E contains only one constraint C', i.e. $E = \{C'\}$. Suppose C' has a scope of S'. We have

$$\begin{aligned} C_\varnothing \oplus \bigoplus_{C_S \in \mathcal{C}} C_S(l[S]) &= C_\varnothing \oplus C'(l[S']) \oplus \bigoplus_{C_S \in \mathcal{C}-E} C_S(l[S]) \\ &\geq C_\varnothing \oplus \bigoplus_{C_S \in \mathcal{C}-E} C_S(l[S]) \end{aligned}$$

for all possible complete assignments l. Note that we can re-write the definition of A-costs as:

$$\text{A-cost}(\mathcal{P}) = \underset{v_1 \in D_1}{Q_1} \underset{v_2 \in D_2}{Q_2} \dots \underset{v_n \in D_n}{Q_n} [C_\varnothing \oplus \bigoplus_{C_S \in \mathcal{C}} C_S(l[S])]$$

where $l = \{x_1 = v_1, x_2 = v_2, \dots, x_n = v_n\}$. In MWCSPs, all quantifiers are either $\min$ or $\max$ which are monotonic aggregators/functions. By monotonic properties, this allow us to achieve,

$$\begin{aligned} &\underset{v_1 \in D_1}{Q_1} \underset{v_2 \in D_2}{Q_2} \dots \underset{v_n \in D_n}{Q_n} [C_\varnothing \oplus \bigoplus_{C_S \in \mathcal{C}} C_S(l[S])] \\ \geq &\underset{v_1 \in D_1}{Q_1} \underset{v_2 \in D_2}{Q_2} \dots \underset{v_n \in D_n}{Q_n} [C_\varnothing \oplus \bigoplus_{C_S \in \mathcal{C}-E} C_S(l[S])] \end{aligned}$$

Observe that $C_\varnothing \oplus \bigoplus_{C_S \in \mathcal{C}-E} C_S(l[S])$ is the cost function for $\mathcal{P}'$, this gives

$$\text{A-cost}(\mathcal{P}) \geq \text{A-cost}(P')$$

where E contains a constraint (i.e. $|E| = 1$). We now have established the lemma for removing one constraint from the problem. In general E has more than one constraint. Since removing multiple constraints can be viewed as removing a series of constraints, and therefore, the lemma holds. □

Theorem 2 *Function $nc_{lb}(\mathcal{P}, x_i = v)$ is a lower bound approximating function* $\text{lbaf}(\mathcal{P}, x_i = v)$. *[15]*

Proof Lemma 3 suggests the computation of A-costs for unary MWCSPs can be done in O(nd), where n is the number of variables and d is the maximum domain size. Therefore, computing the A-costs for any sub-problems is also efficient. The function $nc_{lb}()$ can be seen as a function extracting A-costs for the sub-problem in S with minimal A-costs following Lemma 3, by partitioning unary constraints into three groups: (a) C_j where $j < i$; (b) C_i; and (c) C_j where $j > i$.

(a) 2nd term ($\bigoplus_{j:j<i} \min C_j$): $C_1, C_2, \dots, C_{i-1}$
Recall each sub-problem in the set S will be obtained from $\mathcal{P}$ by assigning a combination of assignments from D_1, D_2, …, D_{i-1}. We should consider the sub-problem in the set having the lowest costs. Therefore, we choose the minimum costs for these unary constraints.

(b) 3rd term ($C_i(v)$): C_i
All sub-problems in the set share a common assignment $x_i = v$, and therefore, we include the costs $C_i(v)$.

(c) 4th term ($\bigoplus_{j:i<j} Q_j C_j$): $C_{i+1}, C_{i+2}, \dots, C_n$
For the max player, he/she could have a strategy which selects values maximizing the unary constraint on his/her variables. On the other hand, whatever values the min player is choosing, the min player must at least incur a cost which is equal to the summation of the minimum costs of the unary constraints on the $\min$ variables. Therefore, this would guarantee the max player must be able to achieve a cost of at least $\bigoplus_{j:i<j} Q_j C_j$.

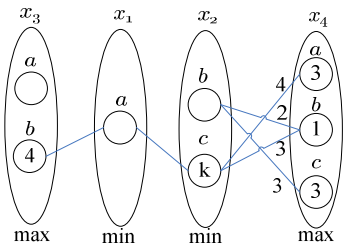

Fig. 8 Constraints for Example 4

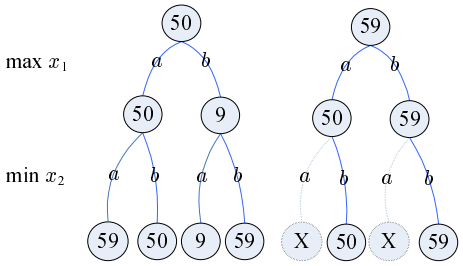

Fig. 9 Example 5: Before and after pruning a of x_2

If $\mathcal{P}$ has only unary constraints, we can observe that function $nc_{lb}()$ will always give a tight lower bound approximation for the set of problems. Note that MWCSPs may have binary constraints or even high-arity constraints. However, Lemma 4 shows that we can infer the lower bound of a problem by estimating the lower bound of a sub-problem with parts of the constraints being ignored. Since $nc_{lb}()$ is a lower bound approximation function for the sub-problem containing unary constraints only, it will also be a lower bound approximation function for general MWCSPs. □

Example 4 We re-use Example 1. Suppose we were at sub-problem $\mathcal{P}' = \mathcal{P}[x_1 = a]$ and we have just visited the further sub-problem $\mathcal{P}'[x_2 = a]$ which has a new upper bound of 7. Before visiting $\mathcal{P}'[x_2 = b]$, we try to prune some values according to Table 1 using the new upper bound. Note that since x_1 is assigned, we assume the solver will reduce all binary constraints constraining on variable x_1 to unary constraints and merge with existing unary constraints. Figure 8 shows the sub-problem $\mathcal{P}'$. Suppose now $nc_{lb}()$ is applied and no unary costs for bounded variables, i.e. $C_\varnothing = 0$. We want to check if the value b can be pruned from D_2. In the sub-problem $\mathcal{P}'[x_2 = b]$, the quantifier Q_3 and Q_4 are both $\max$, and they will take at least the maximum unary cost $\max C_3$ and $\max C_4$. We have $C_\varnothing + \min C_1 + C_2(b) + \max C_3 + \max C_4 = 0 + 0 + 0 + 4 + 3 = 7 \geq ub$. The cost of any ultra-weak solution assignment in the sub-problem $\mathcal{P}'[x_2 = b]$ is at least 7. The value b can therefore be removed from domain D_2. Notice that such a node cannot be pruned by basic alpha-beta pruning.

Some might suggest the following to achieve a tighter bound: 1) replace $i < j$ in the last term by $i \neq j$ to remove the second term, and 2) modify $nc_{lb}(\mathcal{P}, x_i = v)$ such that it returns a lower bound of the A-cost of the sub-problem $\mathcal{P}[x_i = v]$ (instead of a set S of sub-problems). However, these changes may lead to incorrect prunings, as we demonstrate in Example 5.

Example 5 Suppose we were given an MWCSP $\mathcal{P}$ with the ordered sequence of two variables (x_1, x_2), domains $D_1 = D_2 = \{a, b\}$, two unary constraints C_1 and C_2, one binary constraint $C_{1,2}$, and a quantifier sequence $(Q_1 = \max, Q_2 = \min)$. We denote the global upper bound by k. Non-zero costs given by constraints are listed as follows: $C_1(a) = 50$, $C_2(a) = 9$, $C_{1,2}(b, b) = k$. If $k = 59$, the A-cost of $\mathcal{P}$ is 50. Figure 9 (left picture) shows the labeling tree for $\mathcal{P}$. If $i < j$ is replaced by $i \neq j$ in Definition 1, $nc_{lb}(\mathcal{P}, x_2 = a)$ returns $59 >$ A-cost$(\mathcal{P}[x_1 = b][x_2 = a])$ =

```
function computeArrayOfMinCosts(P):
  M[1] = 0
  for i in 2..n:
    j = i - 1
    M[i] = M[j] + min(Cj)
  return M
function computeArrayOfQuantifiedCosts(P):
  Q[n] = 0
  for i in n-1..1:
    j = i + 1
    Q[i] = Q[j] + Q(Cj)
  return Q
function NC_LB(P, xi = v):
  return C∅ + M[i] + Ci(v) + Q[i]
```

Fig. 10 Algorithms for implementing $nc_{lb}()$

9. If $nc_{lb}(\mathcal{P}, x_2 = a)$ returns a lower bound approximation of A-cost$(\mathcal{P}[x_2 = a])$, it may return 59. Both cases may cause value a of x_2 to be pruned. The right picture in Figure 9 shows the labeling tree after pruning, and we can easily observe that the A-cost of the new problem changes from 50 to 59. The pruning is unsound.

We show the function `NC_LB(P, xi = v)` for implementing $nc_{lb}()$ in Figure 10. A direct approach in computing $nc_{lb}()$ is to directly compute all the terms in the function for each of the different assignments $x_i = v$. However, it is easy to note some of these terms, e.g. $\bigoplus_{j:j<i} \min C_j$ and $\bigoplus_{j:i<j} Q_j C_j$, can be pre-computed to avoid unnecessary re-computations for each of the different assignments $x_i = v$. In addition, it is easy to observe that after computing $\bigoplus_{j:j<i} \min C_j$ for a variable x_i, computing the same term for the next variable x_{i+1} is essentially: $(\bigoplus_{j:j<i} \min C_j) + \min C_i$. That means we only need to compute $\min C_i$ and add up the previous result to compute the term for the next variable. Similar approach could be used for the term $\bigoplus_{j:i<j} Q_j C_j$. For each variable x_i, we will compute and maintain the term $\bigoplus_{j:j<i} \min C_j$ and the term $\bigoplus_{j:i<j} Q_j C_j$ by calling function `computeArrayOfMinCosts()` and function `computeArrayOfQuantifiedCosts()` respectively. Array `M` and `Q` will then store the results for these terms respectively. Both functions run in O(nd), where n is the number of variables and d is the maximum variable domain size. After preprocessing the two arrays `M` and `Q`, each query to the function `NC_LB()` to obtain the lower bound runs only in O(c), where c is the constant time. In total, querying `NC_LB()` for all variables and their values in an MWCSP will have a runtime of O(nd). To ease our implementation effort, we implement the $\oplus$ operator using standard addition and the $\ominus$ operator using standard subtraction in all our consistency implementations. We add extra routine to check all addition operations to see if the summed value is larger than k and require re-assigning to k.

5.2 Arc Consistency: Lower Bound

To obtain stronger lower bound, we further define function $ac_{lb}()$ based on $nc_{lb}()$. Without loss of generality, we restrict our attention to MWCSPs which have *only unary constraints and one binary constraint*. We will show that computing any sub-problems for these MWCSPs are efficient (polynomial time), and therefore, computing the lower bound for these sub-problems are again efficient. By similar argument for justifying $nc_{lb}()$, viewing unary constraints plus one binary constraint on general MWCSPs, the bound is still correct.

Definition 2 The $ac_{lb}[C_{i,j}](\mathcal{P}, x_i = v)$ function approximates the A-cost for the set S of sub-problems $\{\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v] | v_1 \in D_1, \ldots, v_{i-1} \in D_{i-1}\}$. Define

$$ac_{lb}[C_{i,j}](\mathcal{P}, x_i = v) \equiv C_{\varnothing} \oplus (\bigoplus_{k:k<i} \min C_k) \oplus (C_i(v)) \\ \oplus (\bigoplus_{k:i<k \wedge j \neq k} Q_k C_k) \oplus (\underset{u \in D_j}{Q_j} \{C_j(u) \oplus C_{i,j}(v,u)\})$$

where $C_{i,j}$ is a binary constraint on variable x_i and x_j where $i < j$, $Q_j \in \mathcal{Q}$ is the quantifier for variable x_j, and $Q_k \in Q$ is the quantifier for variable x_k where $k > i$ and $k \neq j$.

Comparing to $nc_{lb}()$, the first three terms are the same. The fourth term is equivalent to the last term in $nc_{lb}()$, except we do not consider costs for constraint C_j, which will be considered in the fifth term.

Theorem 3 *The function $ac_{lb}[C_{i,j}](\mathcal{P}, x_i = v)$ for binary constraint $C_{i,j}$ is a lower bound approximating function* $\mathrm{lbaf}(\mathcal{P}, x_i = v)$. *[15]*

To prove the function is a lower bound approximation function, we first show Lemma 5.

Lemma 5 *The* A-cost *of an MWCSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q}, k)$ with only unary constraints and **one** binary constraint $C_{i,j}$ is equal to*

$$(\bigoplus_{k \in [1\ldots n] \setminus \{i,j\}} \underset{u \in D_k}{Q_k} C_k(u)) \oplus (\underset{u \in D_i}{Q_i} [\underset{v \in D_j}{Q_j} [C_i(u) \oplus C_j(v) \oplus C_{i,j}(u,v)]])$$

where $Q_i, Q_j, Q_k \in \mathcal{Q}$. [15]

Lemma 5 follows directly from the definition of A-costs.

Proof From Lemma 5, suppose we were given an MWCSP $\mathcal{P}$ with only unary constraints and one binary constraint $C_{i,j}$, its sub-problem $\mathcal{P}[x_{1..i} = v_{1..i}]$ with a fixed value assignment $\{x_1 = v_1, \ldots, x_i = v_i\}$ has the following A-costs:

$$\bigoplus_{k:k<i} C_k(v_k) \oplus \bigoplus_{k \in [i+1\ldots n] \setminus \{j\}} \underset{u \in D_k}{Q_k} C_k(u) \oplus \underset{u \in D_j}{Q_j} [C_i(v_i) \oplus C_j(u) \oplus C_{i,j}(v_i, u)]$$

Since $C_i(v_i)$ in the third term is fixed and it is invariant to any value in D_j, we can rewrite the expression as:

$$\bigoplus_{k:k<i} C_k(v_k) \oplus C_i(v_i) \oplus \bigoplus_{k:i<k\wedge k\neq j} \underset{u\in D_k}{Q_k}\ C_k(u) \oplus \underset{u\in D_j}{Q_j}\ [C_j(u) \oplus C_{i,j}(v_i, u)]$$

One point to note is that only the first term in the above expression $\bigoplus_{k:k<i} C_k(v_k)$ is variant towards different values for variables preceding x_i (i.e. variable x_k where $k < i$). Therefore, for the set of sub-problems $\{\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v] | v_1 \in D_1, \ldots, v_{i-1} \in D_{i-1}\}$ which share the common assignment $x_i = v$, we can observe that the sub-problem(s) which has the minimum A-costs should have a value assignment $x_1 = v_1, x_2 = v_2, \ldots, x_{i-1} = v_{i-1}$ s.t. the A-costs is equal to

$$\bigoplus_{k:k<i} \min C_k \oplus C_i(v_i) \oplus \bigoplus_{k:i<k\wedge k\neq j} \underset{u\in D_k}{Q_k}\ C_k(u) \oplus \underset{u\in D_j}{Q_j}\ [C_j(u) \oplus C_{i,j}(v, u)]$$

This gives a tight lower bound for the set S of sub-problems of the original $\mathcal{P}$ which has only unary constraints and one binary constraint $C_{i,j}$. Note that general MWCSPs may have more than one binary constraints and even high-arity constraints. Since $ac_{lb}[C_{i,j}]()$ is a lower bound approximation function for a sub-problem which considers only unary constraints and a binary constraint $C_{i,j}$, by Lemma 4, the function is also a lower bound approximation function for general MWCSPs. □

Example 6 We re-use Example 4 and Figure 8. Recall we are at sub-problem $\mathcal{P}' = \mathcal{P}[x_1 = a]$ and we have already visited the further sub-problem $\mathcal{P}'[x_2 = a]$. Before visiting $\mathcal{P}'[x_2 = b]$, we now try to prune some values according to Table 1 using $ac_{lb}()$. Similarly, we assume there no unary costs for bounded variables, i.e. $C_\varnothing = 0$, and we want to check if the value b can be pruned from D_2. Function $ac_{lb}[C_{2,4}](\mathcal{P}, x_2 = b)$ is equal to:

$$\begin{aligned}
&C_\varnothing \oplus \min C_1 \oplus C_2(b) \oplus \max C_3 \oplus \max_{u\in D_4}\{C_4(u) \oplus C_{2,4}(b, u)\}\\
=&0 \oplus 0 \oplus 0 \oplus 4 \oplus \max\{C_4(a) \oplus C_{2,4}(b, a), C_4(b) \oplus C_{2,4}(b, b), C_4(c) \oplus C_{2,4}(b, c)\}\\
=&4 \oplus \max\{3 \oplus 0, 1 \oplus 2, 3 \oplus 3\}\\
=&4 \oplus 6 = 10
\end{aligned}$$

The cost of any assignment in the sub-problem $\mathcal{P}'[x_2 = b]$ is at least 10. In previous example (Example 4), $nc_{lb}(\mathcal{P}, x_2 = b)$ returns 7. We can see that considering binary constraint is worthwhile as the lower bound estimation is more accurate.

Note that Definition 2 is only *one* possible approach to define a lower bound approximation function for Arc Consistency (AC), following Lemma 5. It is designed in such a way that only **one** binary constraint is used in bounds calculation for costs estimation, and our approach is similar to AC in Quantified CSPs [23,13]. The only trick in computing the function is to combine costs on constraints C_i, C_j, and $C_{i,j}$ efficiently. Some readers might suggest us to directly transfer the costs computed by $ac_{lb}()$ to $C_\varnothing$ or unary constraints. However, we observe that transferring such costs directly may result in binary constraints with negative costs and violates the

```
1  function AC_LB(P, Cij, xi = v):
2    if Qj == min:
3      minimumCost = +∞
4      for u in Dj:
5        if Cj(u) + Cij(v,u) < minimumCost:
6          minimumCost = Cj(u) + Cij(v,u)
7      return C∅ + M[i] + Ci(v) + [Q[i] - Q(Cj)] + minimumCost
8    if Qj == max:
9      maximumCost = -∞
10     for u in Dj:
11       if Cj(u) + Cij(v,u) > maximumCost:
12         maximumCost = Cj(u) + Cij(v,u)
13     return C∅ + M[i] + Ci(v) + [Q[i] - Q(Cj)] + maximumCost
```

Fig. 11 Algorithms for implementing $ac_{lb}()$

cost range for the valuation structure (i.e. $[0..k]$). Unfortunately, this may also violate a basic assumption in (M)WCSPs that the minimum possible costs a constraint is allowed to return is zero, which is fundamental in designing $nc_{lb}()$ and $ac_{lb}()$.

We now show the function `AC_LB()` for computing $ac_{lb}()$ in Figure 11. The function assumes a common assignment $x_i = v$ and a binary constraint $C_{i,j}$ on x_i were given from the input. The computation of the first three terms: $C_{\varnothing}$, $\bigoplus_{k:k<i} \min C_k$, and $C_i(v)$ in $ac_{lb}()$ is the same as $nc_{lb}()$. For the fourth term $\bigoplus_{k:i<k\wedge j\neq k} Q_k C_k$ in $ac_{lb}()$, it is essentially similar to the last term in $nc_{lb}()$. Recall that the last term in $nc_{lb}()$ is pre-computed using the array `Q` and maintained by function `computeArrayOfQuantifiedCosts()` (in Figure 10). To avoid unnecessary computation, we therefore re-use and modify the result stored in `Q` for the computation in $ac_{lb}()$, by `Q[i] - Q(Cj)` in line 7 and 13. Note that the minus operation used to compute `Q[i] - Q(Cj)` is standard subtraction, not $\ominus$ defined in the Weighted CSP framework, to assure that `Q[i] - Q(Cj)` truly computes the required term. The for loops in line 3 to 6 and line 9 to 12 are used to compute the last term in $ac_{lb}()$, which depends on the quantifiers of the variable x_j.

Suppose we were given a binary constraint C_{ij}, it is not hard to check line 3 to line 7 (for $Q_j = \min$) and line 9 to line 13 (for $Q_j = \max$) run in O(d) assuming the array `M` and `Q` have been pre-computed, where d is the maximum variable domain size. Suppose there are e binary constraints and n variables in an MWCSP, and the maximum variable domain size is d. We will need O(ed) queries to call `AC_LB()` for all binary constraints and all variable assignments $x_i = v$. The overall runtime complexity will then be in O(ed^2) which is bounded by O(n^2d^2).

It is natural for us to further ask for stronger/tighter functions which consider more than one binary constraint. Note that in classical local consistency enforcement such as: AC in CSPs [2]; AC* in Weighted CSPs [17]; and (Q)AC [23] in Quantified CSPs, we usually handle one (binary) constraint at a time. Consistency enforcement will be performed many times at each node of the search tree, and considering multiple constraints at a time may cause a huge increase in time complexity. We have to maintain a balance between amount of reasoning at each search node and amount of pruning achieved. There are stronger consistency notions with efficient algorithms

which consider more than one binary constraint, e.g. Max Restricted Path Consistency [11] in CSPs and OSAC [10] in Weighted CSPs/Valued CSPs. Investigations on stronger notions for MWCSPs is an interesting future work. One possibility to enhance ac_{lb} is to consider a subset of constraints that forms a tree, and employ a dynamic programming approach to enforce such stronger consistencies.

5.3 Node and Arc Consistency: Upper Bounds

In linear programming, duality [24,35] provides a standard way to obtain lower bounds (for minimization problems). In fact, the Principle/Theory of Duality [24] suggests that we can convert the original (primal) problem to its dual form, and tackle the problem by using both forms. We can often find techniques in many areas utilizing the Principle. In integer programming, we can obtain the lower bound of the original problem by tackling the Lagrangian dual [35]. In classical CSPs, hidden variable transformation is a technique which is used to transform general CSPs into binary CSPs (i.e. dual problem). The technique reformulates the original problem by expressing each constraint as a variable [29,2] in the dual problem. In Quantified CSPs, dual consistency [6] was defined by creating the dual problem, involving negation of the original constraints. We will now show how to implement upper bound approximation functions $nc_{ub}()$ and $ac_{ub}()$ by using the duality principle in MWCSPs.

5.3.1 Duality of Constraints

One approach to create $nc_{ub}()/ac_{ub}()$ is to utilize the constraint duality property, which is similar to dual consistency [6] in Quantified CSPs. We first define a dual problem of an MWCSP.

Definition 3 Suppose we were given an MWCSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q}, k)$. A *dual problem* of $\mathcal{P}$ is an MWCSP $\mathcal{P}^\dagger = (\mathcal{X}, \mathcal{D}, \mathcal{C}^\dagger, \mathcal{Q}^\dagger, k)$ s.t. for a complete assignment l,

$$C^\dagger_\varnothing \oplus^\dagger \bigoplus_{C^\dagger_s \in \mathcal{C}^\dagger}{}^\dagger C^\dagger_S(l[S]) = -1 \times (C_\varnothing \oplus \bigoplus_{C_S \in \mathcal{C}} C_S(l[S]))$$

The new valuation structure, called negative valuation structure $V^\dagger(k)$, for the dual problem $\mathcal{P}^\dagger$ will be defined as $([-k...0], \oplus^\dagger, \leq)$. The structure contains a set of integers $[-k...0]$ with standard integer ordering $\leq$. Addition $\oplus^\dagger$ is defined by $a \oplus^\dagger b = \max(-k, a+b)$. For any integer a and b where $a \leq b$, subtraction $\ominus^\dagger$ is defined by $a \ominus^\dagger b = a - b$ if $a \neq -k$, and $a \ominus^\dagger b = -k$ if $a = -k$. We also require $Q^\dagger_i = \min$ if $Q_i = \max$, and $Q^\dagger_i = \max$ if $Q_i = \min$ [15]. Note that in the standard definition of valuation structure, the max element should be an annihilator and the min element should be a neutral element. In the negative valuation structure, the max element 0 is now a neutral element and the min element $-k$ is now an annihilator.

We can observe that A-cost$(\mathcal{P}) = -1 \times$ A-cost$(\mathcal{P}^\dagger)$, and a straightforward method to construct the **dual constraints** is to multiply costs for all constraints in the original problem by -1. The valuation structure $V^\dagger(k)$ for the dual problem are

natural extensions from the original problem$V(k)$, by flipping costs from the positive axis to the negative axis. For the rest of the paper, we may abuse notations by dropping the † sign if the context is clear that we are working on the dual problem. In the dual problem definition, we do not require that every constraint in the dual returns exactly -1 times the costs of its corresponding constraint in the original problem. The reason behind this is that the set of constraints which is efficient and effective to estimate bounds in the original problem may not be necessarily efficient and effective to estimate bounds in the dual problem. We allow solvers and algorithms to handle the dual problem, in particular constraint representation, separately as long as the resulting overall A-costs for the original and the dual is related by a multiplication factor of -1. This would also allow future work focusing on consistencies and transformations on dual problems.

We then show how we utilize dual problem(s) to check $\mathrm{ubaf}(\mathcal{P}, x_i = v) \leq lb$ (Condition 2) for an MWCSP $\mathcal{P}$.

Theorem 4 *Suppose we were given an MWCSP $\mathcal{P}$ and its dual problem $\mathcal{P}^\dagger$. Suppose there is a lower bound approximation function* $\mathrm{lbaf}()$. *[15]*

$$\mathrm{lbaf}(\mathcal{P}^\dagger, x_i = v) \geq -1 \times lb \implies \textit{Condition (2)}$$

The proof of Theorem 4 can be easily constructed from the definition. We can also infer that $\mathrm{lbaf}(\mathcal{P}^\dagger, x_i = v) \times -1$ is an upper bound approximation function for the original problem. In fact, the upper bound $ub^\dagger$ (lower bound $lb^\dagger$ resp.) of $\mathcal{P}^\dagger$ is equal to -1 times the lower bound lb (upper bound ub resp.) of $\mathcal{P}$. Therefore, we further define $ub^\dagger = -1 \times lb$, and $lb^\dagger = -1 \times ub$.

Example 7 We re-use Example 4 and the constraint shown in Figure 8. Recall we are at the sub-problem $\mathcal{P}' = \mathcal{P}[x_1 = a]$ and we have already visited $\mathcal{P}'[x_2 = a]$. Figure 12 and 13 show the labeling tree for the sub-problem and its dual problem. It is not hard to check the A-costs for every sub-problem in the original problem is -1 times the A-costs of its corresponding sub-problem in the dual problem. It is also not hard to infer a lower bound (an upper bound resp.) approximation function for a sub-problem in the dual problem is also an upper bound (a lower bound resp.) approximation function for the original problem (and vice versa). Figure 14 shows a possible set of constraints for the dual problem obtained by directly multiplying -1 to the costs return by the constraints in Figure 8.

We will now show how to implement $nc_{ub}()$ and $ac_{ub}()$, via checking the $nc_{lb}()$ and $ac_{lb}()$ for the dual problem. Recall when we define $nc_{lb}()$ and $ac_{lb}()$, we have made the assumption that the minimum possible costs that can be returned by a constraint is zero. If we construct a dual constraint by multiplying -1 to the costs of its corresponding constraint in the original problem, most of the constraints in the dual problem will return costs less than zero. This would mean that if we need to compute a lower bound approximation for the dual problem, we cannot directly re-use $nc_{lb}()$ and $ac_{lb}()$. In addition, this would also hinder us from re-using transformation techniques from the WCSP framework since most of the transformation techniques have

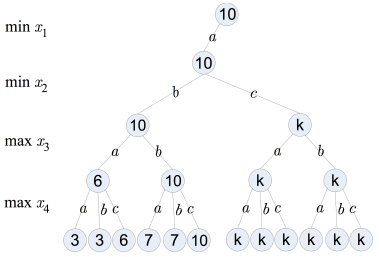

Fig. 12 Labeling tree for the original problem

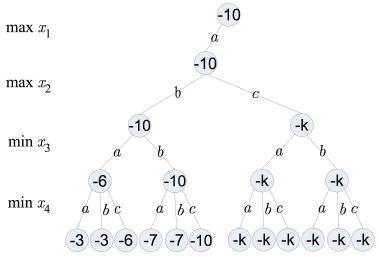

Fig. 13 Labeling tree for the dual problem

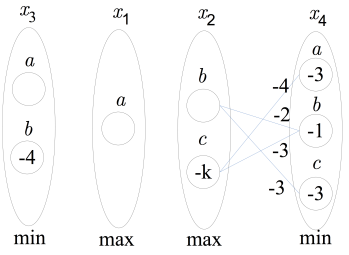

Fig. 14 Dual constraints in Example 7

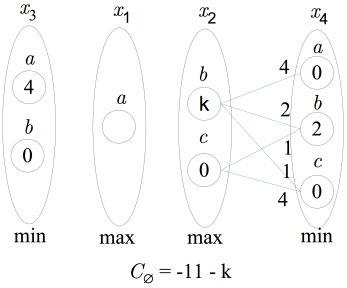

Fig. 15 Normalized constraints in Example 8

the assumption that the minimum possible costs returned by a constraint is zero. To tackle this issue, we further define normalized form for the dual problem to satisfy the assumption.

Definition 4 Suppose we were given a dual problem $\mathcal{P}^\dagger = (\mathcal{X}, \mathcal{D}, \mathcal{C}^\dagger, \mathcal{Q}^\dagger, k)$. The *normalized form* of the dual problem is $\mathcal{P}^N = (\mathcal{X}, \mathcal{D}, \mathcal{C}^N, \mathcal{Q}^\dagger, k)$. We require all constraints except $C^N_\varnothing$ to return non-negative costs, i.e.

$$\forall C^N_s \in \mathcal{C}^N - \{C^N_\varnothing\} : 0 \leq C^N_S(l[S]) \leq k.$$

For all complete assignments l, we have two conditions:

$$[C^\dagger_\varnothing \oplus^\dagger \bigoplus_{C^\dagger_s \in \mathcal{C}^\dagger}{}^\dagger C^\dagger_S(l[S]) = -k] \iff [C^N_\varnothing + \sum_{C^N_s \in \mathcal{C}^N} C^N_S(l[S]) \leq -k], \text{ and}$$

$$[0 \geq C^\dagger_\varnothing \oplus^\dagger \bigoplus_{C^\dagger_s \in \mathcal{C}^\dagger}{}^\dagger C^\dagger_S(l[S]) > -k] \iff$$

$$[C^N_\varnothing + \sum_{C^N_s \in \mathcal{C}^N} C^N_S(l[S]) = C^\dagger_\varnothing \oplus^\dagger \bigoplus_{C^\dagger_s \in \mathcal{C}^\dagger}{}^\dagger C^\dagger_S(l[S])]$$

By observing Definition 4, we can see that all constraints except $C^N_\varnothing$ in the normalized form will return costs from 0 to k. This would allow us to re-use transformation techniques from WCSPs (with slight modification to handle $C^N_\varnothing$). We now give the algorithm to transform a dual problem into its normalized form.

```
1  function Normalize(P):
2  //For unary constraints
3    for Ci in C:
4      minCost = +∞
5      for u in Di:
6       if Ci(u) < minCost: minCost = Ci(u)
7      if minCost < 0:
8        CØ = CØ + minCost
9        for u in Di: Ci(u) = Ci(u) - minCost
10 //For binary constraints
11   for Cij in C:
12     minCost = +∞
13     for u in Di:
14       for v in Dj:
15         if Cij(u,v) < minCost: minCost = Cij(u,v)
16     if minCost < 0:
17       CØ = CØ + minCost
18       for u in Di:
19         for v in Dj:
20           Cij(u,v) = Cij(u,v) - minCost
```

Fig. 16 Algorithms to perform normalization for the dual problem

Lemma 6 *Function* `Normalize` *(in Figure 16) transforms a dual problem into its normalized form.*

Lemma 6 suggests an algorithm to transform a dual problem into its normalized form. The algorithm assumes the dual problem only contains unary and binary constraints. The trick we use to maintain the same overall costs of a complete assignment (except those with costs $\leq -k$) is to allow $C_\varnothing^N$ (the global costs) to become negative and unbounded. The main goal of the function is to transfer costs from $C_\varnothing^\dagger$ to constraints with negative costs until all constraints (except $C_\varnothing^\dagger$) return non-negative costs. It is not hard to see all constraints except $C_\varnothing^\dagger$ will return costs from 0 to k after normalization. We implement and execute the function `Normalize` once during pre-processing in the root node to convert the dual problem into its normalized form. Since the formulations of $nc_{lb}()$ and $ac_{lb}()$ do not require the global costs $C_\varnothing^N$ must be positive, we can re-use $nc_{lb}()$ and $ac_{lb}()$ on the normalized form.

One potential drawback in our definition for normalized form is that we cannot reuse the valuation structure for WCSPs (or the valuation structure for dual problem) to bound costs within $[0..k]$ (or $[-k..0]$). The main reason is that the costs range for $C_\varnothing^N$ could be far less than $-k$ in order to balance the increased costs for the other constraints. A direct consequence is that now k (or $-k$ in the dual form) cannot be used as annihilator. In other words, if k (or $-k$) is found during a sequence of summation operations, we cannot naively conclude the result is k (or $-k$). We have to compute the whole sequence of operations before making any conclusions. To fix this problem, we use standard addition instead of $\oplus$ on the normalized form. This modification may result in costs of a complete assignment equal to $-k$ in the dual problem potentially map to costs less than $-k$ in the normalized form.

Example 8 We use the dual constraints (Figure 14) in Example 7 to illustrate how to create the normalized constraints. To translate a dual constraint into the normalized form, we first compute the minimum possible costs returned by the constraint. If the minimum possible costs m is less than zero (i.e. $m < 0$), we then add $-m$ to all costs returned by the constraints. This will make the constraint return only positive costs (including zero). To maintain costs equivalence, we then subtract $-m$ from $C^N_\varnothing$. Figure 15 shows resulting normalized form for the dual constraints in Figure 14.

Definition 5 and 6 show how we modify nc_{lb} and ac_{lb} to cope with the normalized form. It is not hard to observe we only change $\oplus$ operators to standard additions.

Definition 5 The $nc^N_{lb}(\mathcal{P}^N, x_i = v)$ function approximates the A-cost for a set S of sub-problems $\{\mathcal{P}^N[x_{1..i-1} = v_{1..i-1}, x_i = v] | v_1 \in D_1, \ldots, v_{i-1} \in D_{i-1}\}$. Define

$$nc^N_{lb}(\mathcal{P}^N, x_i = v) \equiv C^N_\varnothing + (\sum_{j:j<i} \min C^N_j) + (C^N_i(v)) + (\sum_{j:i<j} Q^\dagger_j C^N_j)$$

where $Q^\dagger_j \in \mathcal{Q}^\dagger$ is the quantifier in the dual (normalized) problem for variable x_j where $j > i$.

Definition 6 The $ac^N_{lb}[C^N_{i,j}](\mathcal{P}^N, x_i = v)$ function approximates the A-cost for the set S of sub-problems $\{\mathcal{P}^N[x_{1..i-1} = v_{1..i-1}, x_i = v] | v_1 \in D_1, \ldots, v_{i-1} \in D_{i-1}\}$. Define

$$\begin{aligned} ac^N_{lb}[C^N_{i,j}](\mathcal{P}^N, x_i = v) \equiv C^N_\varnothing &+ (\sum_{k:k<i} \min C^N_k) + (C^N_i(v)) \\ &+ (\sum_{k:i<k \wedge j \neq k} Q^\dagger_k C^N_k) + (\min_{u \in D_j} Q^\dagger_j \{C^N_j(u) + C^N_{i,j}(v,u)\}) \end{aligned}$$

where $C^N_{i,j}$ is the binary constraint in the normalized dual problem on variable x_i and x_j where $i < j$, $Q^\dagger_j \in \mathcal{Q}$ is the dual quantifier for variable x_j, and $Q^\dagger_k \in \mathcal{Q}$ is the dual quantifier for variable x_k where $k > i$ and $k \neq j$.

Example 9 Recall in Example 8, Figure 15 shows the resulting normalized form for the dual constraints in Figure 14. Suppose we now apply the new $nc^N_{lb}()$ on the normalized dual problem on the assignment $x_2 = b$, the function will become

$$\begin{aligned} &C^N_\varnothing + \min C^N_1 + C^N_2(b) + \min C^N_3 + \min C^N_4 \\ =&(-11 - k) + 0 + k + 0 + 0 = -11 \end{aligned}$$

It is not hard to check -1 times -11 is a correct upper bound for the sub-problem $\mathcal{P}'[x_2 = b]$ of the original problem. Similarly, suppose we now apply the new $ac^N_{lb}()$ for $C^N_{2,4}$ on the assignment $x_2 = b$, the function will become

$$\begin{aligned} &C^N_\varnothing + \min C^N_1 + C^N_2(b) + \min C^N_3 + \min_{v_4 \in D_4} \{C^N_4(v_4) + C^N_{2,4}(b, v_4)\} \\ =&(-11 - k) + 0 + k + 0 + \min\{4, 4, 1\} = -10 \end{aligned}$$

Note that -1 times -10 is also a correct upper bound for the sub-problem $\mathcal{P}'[x_2 = b]$ of the original problem.

Theorem 5 *The function* $nc_{lb}^{N}(\mathcal{P}^N, x_i = v)$ *is a lower bound approximating function* $\mathrm{lbaf}(\mathcal{P}^N, x_i = v)$. *The function* $ac_{lb}^{N}[C_{i,j}^{N}](\mathcal{P}^N, x_i = v)$ *for binary constraint* $C_{i,j}^{N}$ *is a lower bound approximating function* $\mathrm{lbaf}(\mathcal{P}^N, x_i = v)$.

The proof of Theorem 5 is similar to the proofs of Theorem 2 and 3. We can repeat the same proofs for Theorem 2 and 3 by modifying $\oplus$ operator to standard addition and considering that only $C_{\varnothing}^{N}$ can hold negative costs for a normalized dual problem.

Definition 7 An MWCSP $\mathcal{P}$ is *dual constraint node consistent* (DC-NC) [15] iff:

$$\forall x_i \in \mathcal{X}, \forall v \in D_i : nc_{lb}(\mathcal{P}, x_i = v) < ub, \text{ and}$$
$$\forall x_i \in \mathcal{X}, \forall v \in D_i : nc_{lb}^{N}(\mathcal{P}^N, x_i = v) < ub^{\dagger}$$

Definition 8 An MWCSP $\mathcal{P}$ is *dual constraint arc consistent* (DC-AC) [15] iff:

$$\mathcal{P} \text{ is DC-NC},$$
$$\forall C_{i,j} \in \mathcal{C}, \forall v \in D_i : ac_{lb}[C_{i,j}](\mathcal{P}, x_i = v) < ub, \text{ and}$$
$$\forall C_{ij}^{N} \in \mathcal{C}^N, \forall v \in D_i : ac_{lb}^{N}[C_{ij}^{N}](\mathcal{P}^N, x_i = v) < ub^{\dagger}$$

Since DC-AC requires DC-NC to be satisfied by definition, DC-AC is automatically strictly stronger than DC-NC. Readers may think that the second and third conditions of DC-AC have essentially implied the first condition, and therefore, we can omit the first condition. Assume now we ignore the first condition. For a variable assignment $x_i = v$, if there does not exists any binary constraints $C_{i,j}$ ($C_{i,j}^{N}$ resp.) on x_i, we can see that the second (and third condition resp.) are not going to check the bounds on the assignment as $ac_{lb}()$ ($ac_{lb}^{N}()$ resp.) requires a binary constraint as one of the input. Therefore, we add the first condition to guarantee even if x_i does not have a binary constraint x_i, $nc_{lb}(\mathcal{P}, x_i = v)$ ($nc_{lb}^{N}(\mathcal{P}^N, x_i = v)$ resp.) must still be executed to maintain DC-NC.

Figure 17 shows the algorithm to enforce DC-AC, which can be seen as an implementation of the high-level propagation routine in Figure 6. We skip explaining function `strengthening()` in line 19 and 31 (enclosed in grey boxes), which are used to improve the upper and lower bound estimation functions. This function will be explained in Section 5.4. Function `AC_LB(P, Cjk, xj = u)` (in line 25, defined in Definition 2) and `NC_LB(P, xj = u)` (in line 28, defined in Definition 1) implement the lower bound approximation function `lbaf(P, xj = u)` in the high-level routine (in line 8 of Figure 6). By the duality of constraints, we can implement the upper bound approximation function `ubaf(P, xj = u)` (in line 13 of Figure 6) by re-using the two functions `AC_LB()` and `NC_LB()` (in line 37 and line 40) on the (normalized) dual problem $\mathcal{P}^N$ (by using standard addition instead of $\oplus$ operations). For clarity issues, we abstract the two pruning/backtracking routines into the two functions: `upper_bound_pruning` and `lower_bound_pruning`.

Note that for duality of constraints approach, we have to maintain *both* the original problem $\mathcal{P}$ and the dual problem $\mathcal{P}^N$. If a value is being pruned on the original problem $\mathcal{P}$, we should also prune the value on the dual problem $\mathcal{P}^N$ (and vice versa). In this way, prunings caused by lower bound approximations may tighten upper bound approximations (and vice versa), and triggers more

prunings. To handle this case, we slightly modify the usual pruning/backtracking routines (function `upper_bound_pruning` and `lower_bound_pruning`) by adding line 4 and 11. Note that function `AC_LB` and `NC_LB` will be used to compute bounds for both the original and the dual problem, we have to update the two arrays `M` and `Q` (via calling `computeArrayOfMinCosts` and `computeArrayOfQuantifiedCosts` in line 20 - 21 and line 32 - 33) before using `AC_LB`/`NC_LB`. To enforce DC-NC only, we can just skip calling function `AC_LB` and its corresponding pruning/backtracking routines, by skipping line 25-26 and line 37-38. Note that we cannot skip calling `NC_LB` if we want to maintain DC-AC. The reason behind is that DC-AC by definition requires us to maintain DC-NC and `AC_LB` will not be called for a variable x_i if there are no constraints $C_{i,j}$ on the variable (as `AC_LB` is designed for variables with binary constraints).

We now analyze the time complexity of the propagation algorithm, by first evaluating the while loop from line 16 to line 42. We let c to be the constant time, n to be the number of variables, e to be the number of constraints, and d to be the maximum variable domain size. The two functions `upper_bound_pruning` and `lower_bound_pruning` for handling the pruning and backtracking routines run in O(c). We ignore the function `strengthening()` and assume it runs in unknown time complexity O(s). From previous sections, we know that function `computeArrayOfMinCosts` and `computeArrayOfQuantifiedCosts` run in O(nd) and each query to `NC_LB()` and `AC_LB()` run in O(c) and O(d). From the algorithm, we can see that `NC_LB()` will be queried for each possible value assignments $x_j = u$ from the set of future unassigned variables on both the original problem and the dual problem, its overall running time will be bounded by O(nd). For `AC_LB()`, the function will be queried for all of the possible binary constraints and all possible value assignments again on both the original problem and the dual problem. Therefore, the overall running time for the function will be in O(ed^2). Overall, the time complexity for running the while loop once is: O($s+nd+ed^2$) which is bounded by O($s+n^2d^2$). In case we want to maintain DC-NC only, the runtime for running the loop once is O($s+nd$). In the worst case, we could have the propagation while loop runs for nd times. Therefore, the worst case time complexity for DC-AC is O($snd + n^2d^2 + end^3$) (which is bounded by O($snd + n^3d^3$)) and for DC-NC is O($snd + n^2d^2$).

5.3.2 Duality of Quantifiers

Another way to check Condition (2) for an MWCSP $\mathcal{P}$ is to scrutinize functions implementing $\mathrm{ubaf}(\mathcal{P}, x_i = v)$, by repeating similar reasoning for $nc_{lb}()$ on unary MWCSPs (plus a binary constraint). The idea is to use the duality of quantifiers, by replacing $\min$ quantifiers to $\max$ in the reasoning process. Recall we have three groups of unary constraints to consider. One direct way is to consider the maximum costs, instead of minimum costs from constraints in the first group (group (a)), hence changing quantifiers from $\min$ to $\max$. However, using the resulting upper bound approximation functions, by reasoning on unary MWCSPs is incorrect for general MWCSPs. We cannot neglect costs given by binary constraints (or even higher arity constraints if exists). One way to make the bound correct is to add the maximum costs

```
function upper_bound_pruning(ap_lb,ub):
  if ub <= ap_lb:
    if Qj == min: P = P[xj != u]
                  P^N = P^N[xj != u]
                  changed = true
    if Qj == max: return UB_BTK
function lower_bound_pruning(ap_ub,lb):
  if ap_ub <= lb:
    if Qj == min: return LB_BTK
    if Qj == max: P = P[xj != u]
                  P^N = P^N[xj != u]
                  changed = true
function local_consistency():
  i = firstx(P)
  changed = true
  while changed:
    changed = false

    strengthening(P) *
    computeArrayOfMinCosts(P)
    computeArrayOfQuantifiedCosts(P)
    for j in i..n:
      for u in Dj:
        for Cjk in C:
          ap_lb = AC_LB(P, Cjk, xj = u)
          upper_bound_pruning(ap_lb,ub)

        ap_lb = NC_LB(P, xj = u)
        upper_bound_pruning(ap_lb,ub)

    strengthening(P^N) *
    computeArrayOfMinCosts(P^N)
    computeArrayOfQuantifiedCosts(P^N)
    for j in i..n:
      for u in Dj:
        for C^Njk in C^N:
          ap_ub = -1 × AC_LB(P^N, C^Njk, xj = u)
          lower_bound_pruning(ap_ub,lb)

        ap_ub = -1 × NC_LB(P^N, xj = u)
        lower_bound_pruning(ap_ub,lb)

  return NO_BTK
```

Fig. 17 The propagation routine for using duality of constraints

for constraints which will not be covered in the function. Function $nc_{ub}(\mathcal{P}, x_i = v)$ and $ac_{ub}(\mathcal{P}, x_i = v)$ are given as follows, and we write $\max C^\star$ to mean the maximum costs for constraints which are not considered in the function.

Definition 9 The $nc_{ub}(\mathcal{P}, x_i = v)$ function approximates the A-cost for a set S of sub-problems $\{P[x_{1..i-1} = v_{1..i-1}, x_i = v] | v_1 \in D_1, v_2 \in D_2, \ldots, v_{i-1} \in D_{i-1}\}$.

Define:

$$nc_{ub}(\mathcal{P}, x_i = v) \equiv C_\varnothing \oplus (\bigoplus_{j:j<i} \max C_j) \oplus (C_i(v)) \oplus (\bigoplus_{j:i<j} Q_j C_j) \oplus (\max C^\star)$$

where $Q_j \in \mathcal{Q}$ is the quantifier for x_j, $j > i$.

We can easily observe that $\max C^\star$ is equal to $\bigoplus_{j,k:j\neq k} \max C_{jk}$ if there are only unary and binary constraints.

Definition 10 The function $ac_{ub}[C_{i,j}](\mathcal{P}, x_i = v)$ approximates the A-cost for the set S of sub-problems: $\{P[x_{1..i-1} = v_{1..i-1}, x_i = v] | v_1 \in D_1, v_2 \in D_2, \ldots, v_{i-1} \in D_{i-1}\}$. Define:

$$\begin{aligned} ac_{ub}[C_{i,j}](\mathcal{P}, x_i = v) \equiv C_\varnothing \oplus (\bigoplus_{j:j<i} \max C_j) \oplus (C_i(v)) \oplus (\bigoplus_{k:i<k\wedge j\neq k} Q_k C_k) \\ \oplus Q_j \,{}_{u\in D_j}\{C_j(u) \oplus C_{i,j}(v,u)\} \oplus (\max C^\star) \end{aligned}$$

where $C_{i,j}$ is the binary constraint on x_i and x_j where $i < j$, Q_k is the quantifier for variable x_k where $k > i$ and $k \neq j$, and Q_j is the quantifier for variable x_j.

If there are only unary and binary constraints, $\max C^\star$ is equal to $\bigoplus_{C_{k,l}\in B} \max C_{k,l}$, where $B = \{C_{k,l} \in \mathcal{C} | k \neq l\} - \{C_{i,j}\}$. This paper focuses on efficiently handling unary and binary constraints only. Handling high order constraints and/or global constraints will be left as future work.

Theorem 6 *Function $nc_{ub}(\mathcal{P}, x_i = v)$ is an upper bound approximating function* $\mathrm{ubaf}(\mathcal{P}, x_i = v)$.

Theorem 7 *The function $ac_{ub}[C_{i,j}](\mathcal{P}, x_i = v)$ for binary constraint $C_{i,j}$ is an upper bound approximating function* $\mathrm{ubaf}(\mathcal{P}, x_i = v)$.

To prove the two theorems above, we need to introduce one more lemma.

Lemma 7 *Suppose we were given an MWCSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{Q}, k)$. Let E to be an arbitrary subset of constraints from $\mathcal{C}$ and we define $\mathcal{P}'$ to be an MWCSP obtained from $\mathcal{P}$ by removing all constraints in E (i.e. $\mathcal{P}' = (\mathcal{X}, \mathcal{D}, \mathcal{C} - E, \mathcal{Q}, k)$).*

$$\text{A-cost}(\mathcal{P}) \leq \text{A-cost}(\mathcal{P}') \oplus \bigoplus_{C\in E} \max C$$

where $\max C$ is the maximum possible costs returned by the constraint C.

Proof (Lemma 7) We first consider the simplified case where E contains only one constraint C', i.e. $E = \{C'\}$. Suppose C' has a scope of S'. We have

$$\begin{aligned} C_\varnothing \oplus \bigoplus_{C_S\in\mathcal{C}} C_S(l[S]) &= C_\varnothing \oplus C'(l[S']) \oplus \bigoplus_{C_S\in\mathcal{C}-E} C_S(l[S]) \\ &\leq C_\varnothing \oplus \bigoplus_{C_S\in\mathcal{C}-E} C_S(l[S]) \oplus \max C' \end{aligned}$$

for all possible complete assignments l. Note that we can re-write the definition of A-costs as:

$$\text{A-cost}(\mathcal{P}) = \underset{v_1 \in D_1}{Q_1} \underset{v_2 \in D_2}{Q_2} \dots \underset{v_n \in D_n}{Q_n} [C_\varnothing \oplus \bigoplus_{C_S \in \mathcal{C}} C_S(l[S])]$$

where $l = \{x_1 = v_1, x_2 = v_2, \dots, x_n = v_n\}$. In MWCSPs, all quantifiers are either $\min$ or $\max$ which are monotonic aggregators/functions. By monotonic properties, this allow us to achieve,

$$\begin{aligned}
&\underset{v_1 \in D_1}{Q_1} \underset{v_2 \in D_2}{Q_2} \dots \underset{v_n \in D_n}{Q_n} [C_\varnothing \oplus \bigoplus_{C_S \in \mathcal{C}} C_S(l[S])] \\
\leq &\underset{v_1 \in D_1}{Q_1} \underset{v_2 \in D_2}{Q_2} \dots \underset{v_n \in D_n}{Q_n} [C_\varnothing \oplus \bigoplus_{C_S \in \mathcal{C}-E} C_S(l[S]) \oplus \max C'] \\
= &\max C' \oplus \underset{v_1 \in D_1}{Q_1} \underset{v_2 \in D_2}{Q_2} \dots \underset{v_n \in D_n}{Q_n} [C_\varnothing \oplus \bigoplus_{C_S \in \mathcal{C}-E} C_S(l[S])]
\end{aligned}$$

Observe that $C_\varnothing \oplus \bigoplus_{C_S \in \mathcal{C}-E} C_S(l[S])$ is the cost function for $\mathcal{P}'$, this gives

$$\text{A-cost}(\mathcal{P}) \leq \max C' \oplus \text{A-cost}(P')$$

where E contains a constraint (i.e. $|E| = 1$). We now have established the lemma for removing one constraint from the problem. In general E has more than one constraint, however, removing multiple constraints can be viewed as removing a series of constraints and the lemma therefore holds. □

Proof (Theorems 6 and 7) To prove the two functions $nc_{ub}()$ and $ac_{ub}()$ are correct upper bound approximation functions, we re-use Lemma 3 and 5 again. Suppose we were given an MWCSP $\mathcal{P}$ with only unary constraints (unary constraints and one binary constraint $C_{i,j}$ resp.), its sub-problem $\mathcal{P}[x_{1..i} = v_{1..i}]$ with a fixed value assignment $\{x_1 = v_1, \dots, x_i = v_i\}$ has the following A-costs:

$$\bigoplus_{j:j<i} C_j(v_j) \oplus C_i(v_i) \oplus \bigoplus_{j:j>i} Q_j C_j$$

$$(\bigoplus_{k:k<i} C_k(v_k) \oplus \bigoplus_{k \in [i+1 \dots n] \setminus \{j\}} Q_k\, C_k \oplus \underset{u \in D_j}{Q_j} [C_i(v_i) \oplus C_j(u) \oplus C_{i,j}(v_i, u)] \text{ resp.})$$

Only the first term in the above expression $\bigoplus_{j:j<i} C_j(v_j)$ ($\bigoplus_{k:k<i} C_k(v_k)$ resp.) is variant towards different values being assigned for variables preceding x_i. Therefore, for the set of sub-problems $\{\mathcal{P}[x_{1..i-1} = v_{1..i-1}, x_i = v] | v_1 \in D_1, \dots, v_{i-1} \in D_{i-1}\}$ which share the common assignment $x_i = v$, we can observe that the sub-problem(s) which has the maximum A-costs should have a value assignment $x_1 = v_1, x_2 = v_2, \dots, x_{i-1} = v_{i-1}$ s.t. the A-costs is equal to:

$$\bigoplus_{j:j<i} \max C_j \oplus C_i(v) \oplus \bigoplus_{j:j>i} Q_j C_j$$

$$(\bigoplus_{k:k<i} \max C_k \oplus C_i(v) \oplus \bigoplus_{k:i<k \wedge k \neq j} Q_k\, C_k \oplus \underset{u \in D_j}{Q_j} [C_j(u) \oplus C_{i,j}(v, u)] \text{ resp.})$$

This gives a tight upper bound for the set S of sub-problems of the original $\mathcal{P}$ which has only unary constraints (unary constraints and one binary constraint resp.). However, we cannot neglect costs given by higher arity constraints. From lemma 7, we know that adding the maximum costs of all constraints which have not been considered (including $C_\varnothing$ which is a constant) will give an upper bound to the problem. This completes the proof. □

Example 10 We again re-use Example 4 and the constraint shown in Figure 8. Recall we are at the sub-problem $\mathcal{P}' = \mathcal{P}[x_1 = a]$ and we have already visited $\mathcal{P}'[x_2 = a]$. Function $nc_{ub}(\mathcal{P}', x_2 = b)$ will be:

$$C_\varnothing \oplus \max C_1 \oplus C_2(b) \oplus \max C_3 \oplus \max C_4 \oplus \max C_{2,4}$$

which is equal to 11. Function $ac_{ub}[C_{2,4}](\mathcal{P}', x_2 = b)$ will be:

$$\begin{aligned}
&C_\varnothing \oplus \max C_1 \oplus C_2(b) \oplus \max C_3 \oplus \max_{u \in D_4}\{C_4(u) \oplus C_{2,4}(b,u)\} \\
=&0 \oplus 0 \oplus 0 \oplus 4 \oplus \max\{C_4(a) \oplus C_{2,4}(b,a), C_4(b) \oplus C_{2,4}(b,b), C_4(c) \oplus C_{2,4}(b,c)\} \\
=&4 \oplus \max\{3,3,6\} = 10
\end{aligned}$$

By observing the labeling tree in Figure 12, the two functions return correct lower bound approximation for $\mathcal{P}'[x_2 = b]$.

Note that in general MWCSPs, we may have high-arity constraints and/or global constraints. Computing $\max C^\star$ for $nc_{ub}()/ac_{ub}()$ precisely during search essentially means we have to find the maximum costs for each of these constraints, which could be extremely computational expensive. One naive way to deal with high-arity and/or global constraints is to pre-compute or estimate these maximum costs only once during pre-processing in the root node. We then re-use these pre-computed costs during search. In this paper, we deal with unary and binary constraints *only*. We will maintain and update the maximum costs of all unary and binary constraints during search. Efficient methods to compute and estimate the maximum costs for high-arity / global constraints during search will be left as future works.

Figure 18 shows the algorithm to compute $nc_{ub}()$, which is similar to the algorithm for computing $nc_{lb}()$. We again pre-compute the second and fourth term to avoid unnecessary re-computations for different assignments $x_i = v$. The second term $\bigoplus_{j:j<i} \max C_j$ is pre-computed using function `computeArrayOfMaxCosts`, and results are stored in the array `N`. It is easy to see `computeArrayOfMaxCosts` is similar to `computeArrayOfMinCosts` for computing $nc_{lb}()$ and the time complexity is in O(nd), where n is the number of variables and d is the maximum variable domain size. We re-use the function `computeArrayOfQuantifiedCosts` in Figure 10 to compute the fourth term in $nc_{ub}()$. We use function `computeBinaryMaxCosts` to sum up all the maximum costs for all binary constraints. It is worth noting that computing `computeBinaryMaxCosts` essentially scans *all tuples* of *all* binary constraints, and can be extremely time expensive to compute (especially if it is used frequently). The time complexity in the worst case is in O(ed^2), where e is the number of binary constraints in the

```
1  function computeArrayOfMaxCosts(P):
2    N[1] = 0
3    for i in 2..n:
4      j = i - 1
5      N[i] = N[j] + max(Cj)
6    return N
7  function max(Cij):
8    maxCost = -∞
9    for u in Di:
10     for v in Dj:
11       if maxCost < Cij(u,v): maxCost = Cij(u,v)
12   return maxCost
13 function computeBinaryMaxCosts(P):
14   totalSum = 0
15   for Cij in C: totalSum += max(Cij)
16   return totalSum
17 function NC_UB(P, xi = v):
18   return C∅ + N[i] + Ci(v) + Q[i] + totalSum
```

Fig. 18 Algorithms for implementing $nc_{ub}()$

problem and d is the maximum variable domain size. One way to improve the efficiency in our solver is to compute the maximum costs of all binary constraints and their total sum `totalSum` during pre-processing and then maintain these costs (including `totalSum`) in the global memory. We will need O(e) space where e is the number of binary constraints. We only perform update when a binary constraint is removed/modified, e.g. values being pruned / value assignments occur. After the modification, function `max(Cij)` will immediately return costs from the memory instead of computing costs by scanning tuples from the binary constraint. Function `computeBinaryMaxCosts(P)` will also immediately return the total maximum cost `totalSum` from the global memory. This could reduce the worst case complexity of computeBinaryMaxCosts(P) to O(c), where c is the constant time if the global memory is well maintained. If all of the required functions have been pre-computed, `NC_UB(P, xi = v)` runs in constant time.

We now show the function `AC_UB()` for computing $ac_{ub}()$ in Figure 19, which is similar to function `AC_LB()` in Figure 11. The function again assumes a common assignment $x_i = v$ and a binary constraint $C_{i,j}$ on x_i were given from the input. The computation of the first three terms: $C_\varnothing$, $\bigoplus_{j:j<i} \max C_j$, and $C_i(v)$ in $ac_{ub}()$ is the same as $nc_{ub}()$. The fourth term and the fifth term are essentially the same as in $ac_{lb}()$, and therefore, we adopt the same routine. The only major new requirement in computing $ac_{ub}()$ is to compute the last term, which is the maximum costs of all constraints which are not considered in the function. We compute the term (in line 2) by utilizing again the two functions `computeBinaryMaxCosts(P)` and `max(Cij)` for the computations of $nc_{lb}()$.

Computing `maxC`* in line 2 will use constant time as `max(Cij)` returns the maximum costs directly from the maintained global memory. Similar to `AC_LB()`, line 4 to line 8 (for $Q_j = \min$) and line 10 to line 14 (for $Q_j = \max$) run in O(d), where d is the maximum variable domain size.

```
1  function AC_UB(P, Cij, xi = v):
2    maxC* = totalSum - max(Cij)
3    if Qj == min:
4      minimumCost = +∞
5      for u in Dj:
6        if Cj(u) + Cij(v,u) < minimumCost:
7          minimumCost = Cj(u) + Cij(v,u)
8      return C∅ + N[i] + Ci(v) + [Q[i] - Q(Cj)] + minimumCost + maxC*
9    if Qj == max:
10     maximumCost = -∞
11     for u in Dj:
12       if Cj(u) + Cij(v,u) > maximumCost:
13         maximumCost = Cj(u) + Cij(v,u)
14     return C∅ + N[i] + Ci(v) + [Q[i] - Q(Cj)] + maximumCost + maxC*
```

Fig. 19 Algorithms for implementing $ac_{ub}()$

We now define the node and arc consistencies by utilizing the constructed functions.

Definition 11 An MWCSP $\mathcal{P}$ is *dual quantifier node consistent* (DQ-NC) [15] iff:

$$\forall x_i \in \mathcal{X}, \forall v \in D_i : nc_{lb}(\mathcal{P}, x_i = v) < ub, \text{ and}$$
$$\forall x_i \in \mathcal{X}, \forall v \in D_i : nc_{ub}(\mathcal{P}, x_i = v) > lb$$

Definition 12 An MWCSP $\mathcal{P}$ is *dual quantifier arc consistent* (DQ-AC) [15] iff:

$$\mathcal{P} \text{ is DQ-NC},$$
$$\forall C_{i,j} \in \mathcal{C}, \forall v \in D_i : ac_{lb}[C_{i,j}](\mathcal{P}, x_i = v) < ub, \text{ and}$$
$$\forall C_{i,j} \in \mathcal{C}, \forall v \in D_i : ac_{ub}[C_{i,j}](\mathcal{P}, x_i = v) > lb$$

Since DQ-AC requires DQ-NC to be satisfied by definition, DQ-AC is automatically strictly stronger than DQ-NC.

We show the algorithm to enforce DQ-NC/AC in Figure 20, which is similar to the algorithm for DC-NC/AC. Similarly, we skip explaining function `strengthening()` in line 17 and 29 (enclosed in the grey boxes), which is used to improve the upper and lower bound estimation functions. The function will be explained in Section 5.4. The algorithm can be seen as another implementation of the high-level propagation routine in Figure 6. Similar to the previous algorithm for enforcing DC-NC/AC, we abstract the two pruning/backtracking routines into the two functions: `upper_bound_pruning` and `lower_bound_pruning`. For the implementation of the lower bound estimation function `lbaf(P, xj = u)` (in line 8 of Figure 6), the enforcing algorithm for DQ-NC/AC is the same as DC-NC/AC. The only major difference is on the implementation of the upper bound estimation function `ubaf(P, xj = u)` (in line 13 of Figure 6), where DQ-NC/AC directly implements `ubaf(P, xj = u)` by using function `AC_UB` (in line 36) and `NC_UB` (in line 39). To enforce DQ-NC only, we skip calling function `AC_LB` and function `AC_UB` (and their corresponding pruning/backtracking routines), by skipping

```
1  function upper_bound_pruning(ap_lb,ub):
2    if ub <= ap_lb:
3      if Qj == min: P = P[xj != u]
4                    changed = true
5      if Qj == max: return UB_BTK
6  function lower_bound_pruning(ap_ub,lb):
7    if ap_ub <= lb:
8      if Qj == min: return LB_BTK
9      if Qj == max: P = P[xj != u]
10                   changed = true
11 function local_consistency():
12   i = firstx(P)
13   changed = true
14   while changed:
15     changed = false
16
17     strengthening(P) *
18     computeArrayOfMinCosts(P)
19     computeArrayOfQuantifiedCosts(P)
20     for j in i..n:
21       for u in Dj:
22         for Cjk in C:
23           ap_lb = AC_LB(P, Cjk, xj = u)
24           upper_bound_pruning(ap_lb,ub)
25
26         ap_lb = NC_LB(P, xj = u)
27         upper_bound_pruning(ap_lb,ub)
28
29     strengthening(P) *
30     computeArrayOfMaxCosts(P)
31     computeArrayOfQuantifiedCosts(P)
32     computeBinaryMaxCosts(P)
33     for j in i..n:
34       for u in Dj:
35         for Cjk in C:
36           ap_ub = AC_UB(P, Cjk, xj = u)
37           lower_bound_pruning(ap_ub,lb)
38
39         ap_ub = NC_UB(P, xj = u)
40         lower_bound_pruning(ap_ub,lb)
41
42   return NO_BTK
```

Fig. 20 The propagation routine for using duality of quantifiers

line 23-24 and line 36-37. Similar to the enforcing algorithm for DC-AC (with similar reasons), we cannot skip calling `NC_LB` / `NC_UB` if we want to maintain DQ-AC.

We now analyze the time complexity of the propagation algorithm, by first evaluating the while loop from line 14 to line 41. We let c to be the constant time, n to be the number of variables, e to be the number of constraints, and d to be the maximum variable domain size. The two function `upper_bound_pruning` and `lower_bound_pruning` for handling the pruning and backtracking routines run in O(c). We ignore the function `strengthening()` and assume it

runs in unknown time complexity O(s). Function `computeArrayOfMinCosts`, `computeArrayOfQuantifiedCosts`, and `computeArrayOfMaxCosts` run in O(nd). If function `computeBinaryMaxCosts(P)` in line 32 computes the maximum costs of all binary constraints and their summation `totalSum` via scanning all tuples of all binary constraints, the function will run in the worst case O(ed^2). Note that in practice, we could reduce the complexity of `computeBinaryMaxCosts(P)` by updating the maximum costs of binary constraints incrementally. We only need to update binary constraints which have a variable with values being pruned/assigned. Each query to `NC_LB()` and `NC_UB()` run in O(c) and each query to `AC_LB()` and `AC_UB()` run in O(d). From the algorithm, we can see that `NC_LB()` and `NC_UB()` will be queried for each possible value assignments $x_j = u$ from the set of future unassigned variables. Their overall running time will be bounded by O(nd). For `AC_LB()` and `AC_UB()`, the function will be queried for all of the possible binary constraints and all possible value assignments. The overall running time for the function will be in O(ed^2). Overall to maintain DQ-AC, the time complexity for running the while loop once is: O($s+nd+ed^2$) which is bounded by O($s+n^2d^2$). In the worst case, we could have the propagation while loop runs for nd times. Therefore, the worst case time complexity is O($snd+n^2d^2+end^3$) which is bounded by O($snd + n^3d^3$). If we want to maintain DQ-NC, the time complexity for running the while loop once is still O($s + nd + ed^2$) due to the requirement to compute `computeBinaryMaxCosts(P)`. Suppose we now choose to update the maximum costs of binary constraints (and their total sum) incrementally. If there are r binary constraints constraining on variables with values being pruned, `computeBinaryMaxCosts(P)` would run in O(rd^2). The time complexity to maintain DQ-NC for running the loop once would be changed to O($s+nd+rd^2$). In the worst case, we could have the propagation while loop runs for nd times. A naive calculation for the worst case complexity would give O($snd + n^2d^2 + rnd^3$), where r is an average number of binary constraints need to be updated per execution of the while loop.

5.4 Strengthening Consistencies by Projection/Extension

Consistency algorithms for Weighted CSPs use an equivalence preserving transformation called *projection* [9] to move costs from higher arity constraints to lower arity ones to extract and store bound information. Some of these consistency algorithms also use *extension* [9], which is the inverse of projection, to increase the consistency strength. We propose to re-use Weighted CSP consistencies, especially the parts related to projections and extensions, to strengthen [15] the approximating functions for MWCSPs. We will first give an introduction for projections before showing how we utilize the consistencies for MWCSPs.

Suppose now we have a constraint C_S on the set S of variables, and the costs incurred by C_S is at least c for any assignments on C_S. We can easily infer C_S must be (at least) giving a cost of c. We can extract c from C_S to $C_\varnothing$. The operation performing the extraction is called *0-projections* [36], which is an operation transforming

(C_S, $C_\varnothing$) to (C'_S,$C'_\varnothing$) by projecting a cost of c s.t.:

$$C'_S(l[S]) = C_S(l[S]) \ominus c$$
$$C'_\varnothing = C_\varnothing \oplus c$$

for all possible complete assignments l. Similarly, if we have a constraint C_S on the set S of variables, and the costs incurred by C_S is at least c for any assignments l where $x_i = a \in l$ on C_S, we can extract c from $C_S[l]$ to $C_i(a)$. The operation performing the extraction is called *1-projections* [36], which is an operation transforming (C_S, $C_i(a)$) to (C'_S,$C'_i(a)$) by projecting a cost of c s.t.:

$$\begin{aligned} C'_S(l[S]) &= C_S(l[S]) \ominus c, && \text{if } x_i = a \in l \\ &= C_S(l[S]), && \text{if } x_i = a \notin l \\ C'_i(v) &= C_i(v) \oplus c, && \text{if } v = a \\ &= C_i(v), && \text{if } v \neq a \end{aligned}$$

for all possible complete assignments l. Recall function $nc_{lb}()$ extracts minimum costs from unary constraints. Performing 0-projections on unary constraints before computing $nc_{lb}()$ helps us to pre-compute these minimum costs. Furthermore, by using 1-projections on binary constraints, we may further strengthen $nc_{lb}()$ as costs from binary constraints are being transferred to unary constraints. In function ac_{lb}, not *all* binary constraints are taken into account. If we are able to extract costs from binary constraints to unary constraints by 1-projections, we could also make ac_{lb} a tighter bound. Note that we can also define general k-projections where k is an arbitrary number. However in practice, consistency algorithms usually focus on utilizing 0-/1-projections.

Weighted CSPs consistencies consist of two kinds of conditions: one for pruning and one for projection/extension. However, the general pruning conditions in WCSPs are unsound with respect to MWCSPs. In WCSPs, if an assignment $x_i = v$ has a cost of k, we can prune the assignment. However in MWCSPs, we cannot directly prune value v. We have to further consider quantifier information. If $Q_i = \max$ and we prune the value, the pruning may change the overall A-costs of the problem. Instead of pruning the value, we should perform backtrack (or prune all values to trigger backtrack in QCSPs) according to Table 1. Therefore, we adopt only their projection/extension conditions so as to strengthen DC-NC/AC and DQ-NC/AC. The projection/extension conditions for NC*, AC*, and FDAC* [17, 16] are as follows:

proj-NC*: $\forall C_i, \exists v \in D_i : C_i(v) = 0$

proj-AC*: proj-NC*, and

$\forall C_{i,j}, \forall v_i \in D_i, \exists v_j \in D_j : C_{i,j}(v_i, v_j) = 0$, and

$\forall C_{i,j}, \forall v_j \in D_j, \exists v_i \in D_i : C_{i,j}(v_i, v_j) = 0$

proj-FDAC*: proj-AC*, and

$\forall C_{i,j} : i < j, \forall v_i \in D_i, \exists v_j \in D_j : C_{i,j}(v_i, v_j) \oplus C_j(v_j) = 0$

The main idea of these projection conditions is to require certain tuples of unary and/or binary constraints after transformation by using consistency algorithms have

a cost of 0, i.e. the minimum costs in Weighted CSPs. To satisfy the requirements, consistency algorithms in Weighted CSPs will utilize projections (and further extensions which are the reverse of projections for proj-FDAC*) to transfer costs from higher arity constraints to lower arity constraints. Note that in terms of definition, proj-FDAC* has stronger requirements/conditions than proj-AC* and proj-AC* has stronger requirements/conditions than proj-NC*. Similar to classical CSPs, the enforcement algorithm for enforcing stronger consistencies will usually run slower.

The enforcing algorithm to enforce these conditions have long been developed and discussed in the Weighted CSP community. In this section, we will discuss on how to adopt the enforcing algorithms for proj-NC*, proj-AC*, and proj-FDAC* from the Weighted CSP framework in our implementations. Figure 21, 22, and 23 show three functions: `PROJ-NC⋆`, `PROJ-AC⋆`, and `PROJ-FDAC⋆`, which are used to enforce proj-NC*, proj-AC*, and proj-FDAC* respectively. When enforcing DC-NC/AC and DQ-NC/AC, the appropriate enforcing algorithms: `PROJ-NC⋆`, `PROJ-AC⋆`, or `PROJ-FDAC⋆` will be selected and invoked via calling function `strengthening` (line 19 and 31 in Figure 17 and line 17 and 29 in Figure 20). Since we directly adopted these algorithms from Weighted CSP framework, we will show how to modify these algorithms to cope with MWCSPs.

```
1 function PROJ-NC⋆(P):
2   for Ci in C:
3     minCost = +∞
4     for u in Di:
5       if Ci(u) < minCost: minCost = Ci(u)
6     C∅ = C∅ + minCost
7     for u in Di:
8       Ci(u) = Ci(u) - minCost
```

Fig. 21 Algorithms to enforce proj-NC*

Function `PROJ-NC⋆` enforces proj-NC* by transferring the minimum costs of unary constraints to $C_\varnothing$ and the function runs in O(nd), where n is the number of variables and d is the maximum domain size. Similarly, function `PROJ-AC⋆` enforces proj-AC* by transferring the minimum costs of binary constraints to unary constraints, and further enforce proj-NC* by calling `PROJ-NC⋆`. The runtime of the function is in O(ed^2), where e is the number of binary constraints in the propagation queue `propQueue` and d is the maximum domain size. For function `PROJ-FDAC⋆`, it first enforces proj-AC* by calling function `PROJ-AC⋆`. Then, it maintains proj-FDAC* by a series of extensions (from unary constraints) and projections (from binary constraints). The runtime of the algorithm is again in O(ed^2), where e is the number of binary constraints in the two propagation queues and d is the maximum domain size. Note that the above time complexity estimation assumes the procedure to find and add binary constraints to `propFDACQueue` runs in constant time (line 9 and 18 in Figure 22 and line 14 in Figure 23). Instead of checking whether we need to maintain proj-AC*/proj-FDAC* for all binary constraints every time, `PROJ-AC⋆`/`PROJ-FDAC⋆` maintains a propagation

```
1  function PROJ-AC*(P):
2    for Cij in propQueue:
3      for u in Di:
4        minCost = +∞
5        for v in Dj:
6          if Cij(u,v) < minCost: minCost = Cij(u,v)
7        if minCost > 0:
8          if Ci(u) == 0:
9            for k in [1..i-1]:  propFDACQueue.append(Cki)
10         Ci(u) = Ci(u) + minCost
11         for v in Dj: Cij(u,v) = Cij(u,v) - minCost
12     for v in Dj:
13       minCost = +∞
14       for u in Di:
15         if Cij(u,v) < minCost: minCost = Cij(u,v)
16       if minCost > 0:
17         if Cj(v) == 0:
18           for k in [1..j-1]:  propFDACQueue.append(Ckj)
19         Cj(v) = Cj(v) + minCost
20         for u in Di: Cij(u,v) = Cij(u,v) - minCost
21     if DQ==true: updateBinaryMaxCost(Cij) *
22   propQueue = []
23   PROJ-NC*(P)
```

Fig. 22 Algorithms to enforce proj-AC*

queue `propQueue`(in line 2 in Figure 22) / `propFDACQueue`(in line 5 in Figure 23) for storing all binary constraints which may not satisfy proj-AC*/proj-FDAC*. To maintain `propQueue`/`propFDACQueue`, we will need to further modify function `upper_bound_pruning` and `lower_bound_pruning` in both Figure 17 and 20. When a value v for a variable x_i is being pruned in function `upper_bound_pruning`/ `lower_bound_pruning`, all binary constraints related to the variable will be added to the array `propQueue`, i.e. $C_{i,j}, j > i$ and also $C_{j,i}, j < i$. For the array `propFDACQueue`, we will only add binary constraints $C_{j,i}$ where $j < i$.

After enforcing proj-NC* (proj-AC* resp.), the minimum costs of all unary (unary & binary resp.) constraints must be zero. Therefore, it is unnecessary for us to compute the minimum costs for a unary (unary and binary resp.) constraint again in `NC_LB`, `AC_LB`, `NC_UB`, and `AC_UB` if we have already enforced proj-NC* (proj-AC* resp.). We can then further simplify function `computeArrayOfMinCosts` and `computeArrayOfQuantifiedCosts` in Figure 10. Note that after enforcing projection/extension conditions, the maximum costs for a unary/binary constraint may be changed, and we have to re-compute these maximum costs. For unary constraints, the re-computations *always* occur in function `computeArrayOfQuantifiedCosts` (and also `computeArrayOfMaxCosts` for duality of quantifiers routine) after invoking the routine `strengthening`. For binary constraints, their maximum costs will only be used in ac_{ub} for the duality of quantifier approach. Function `updateBinaryMaxCost(Cij)` is added (enclosed in the grey boxes in

```
1  function PROJ-FDAC*(P):
2    PROJ-AC*(P)
3    for j in [n .. 1]:
4      for i in [j-1 .. 1]:
5        if Cij in propFDACQueue:
6          P = []
7          E = []
8          for u in Di:
9            minCost = +∞
10           for v in Dj:
11             if Cij(u,v) + Cj(v) < minCost: minCost = Cij(u,v) + Cj(v)
12           P[u] = minCost
13           if P[u] > 0 and Ci(u) == 0:
14             for k in [1..i-1]:  propFDACQueue.append(Cki)
15         for v in Dj:
16           maxCost = -∞
17           for u in Di:
18             if P[u] - Cij(u,v) > maxCost: maxCost = P[u] - Cij(u,v)
19           E[v] = maxCost
20         for v in Dj:
21           Cj(v) = Cj(v) - E[v]
22           for u in Di: Cij(u,v) = Cij(u,v) + E[v]
23         for u in Di:
24           Ci(u) = Ci(u) + P[u]
25           for v in Dj: Cij(u,v) = Cij(u,v) - P[u]
26         minCost = +∞
27         for u in Di:
28           if Ci(u) < minCost: minCost = Ci(u)
29         C∅ = C∅ + minCost
30         for u in Di: Ci(u) = Ci(u) - minCost
31         if DQ==true: updateBinaryMaxCost(Cij) *
32   propFDACQueue = []
```

Fig. 23 Algorithms to enforce proj-FDAC*

Figure 22 and Figure 23) to update the maximum costs for the binary constraint `Cij`. Note that the function should only be invoked if duality of quantifier approach is used. We add a global flag `DQ` to distinguish the two dualities in our solver, where the flag is set to `true` if duality of quantifier approach is enabled.

Suppose we were given a binary constraint $C_{i,j}$ where x_i is a $\min$ variable and x_j is a $\max$ variable. Notice that when the enforcing algorithm `PROJ-FDAC*` (in Figure 23) enforces proj-FDAC* for $C_{i,j}$, extension operations may transfer unary costs from C_j (line 21) to $C_{i,j}$ and projection operations may transfer unary costs from $C_{i,j}$ to C_i (line 24). Decreasing unary costs for $\max$ variables and at the same time increasing unary costs for $\min$ variables may weaken the approximating functions. Weakening the approximating function nc_{lb} and ac_{lb} may give a looser lower bound while weakening the approximating function nc_{ub} (in the duality of quantifier approach) may give a looser upper bound. Recall that we also utilize nc_{lb} and ac_{lb} for computing upper bounds in the duality of constraint method. Therefore, we also risk computing looser upper bounds.

The reason behind is that proj-FDAC* will also perform extensions, which are the reverse of projections, to transfer costs from unary constraints to binary constraints. If we transfer costs from unary constraints to binary constraints and then further perform projections transferring costs from these binary constraints to other unary constraints, we can see that costs are being transferred between unary constraints (by using binary constraints). Note that the transformation is a cost equivalence preserving operation, i.e. the total costs of a complete assignment remains unchanged. Therefore, incorporating such transformation in our framework does not affect the soundness of our proposed algorithm. However, transferring costs from $\max$ variable to $\min$ variable may weaken the approximation functions we defined for the framework. One way to tackle and ease the issue is to use a different ordering for the variables when enforcing proj-FDAC*, with $\max$ variables first. The idea is that if $\max$ variables are ordered before the $\min$ variables, we can avoid transferring unary costs from $\max$ variables to $\min$ variables. Note that applying the algorithm to enforce proj-FDAC* on a different variable ordering is still costs equivalence preserving (i.e. soundness still holds). To implement the modification, we only need to add a procedure in the root node during preprocessing to create a new ordering to order max variables first. Note that the for-loops in line 3, 4, and 14 in Figure 23, and also line 9 and 18 in Figure 22 will need to be referenced to the newly created ordering. One interesting future work is to devise a variable ordering heuristic for proj-FDAC* which could order the variable dynamically depending on the quantifiers and constraint costs.

We now re-define DC-NC, DC-AC, DQ-NC, and DQ-AC, to allow users to plug in general projection/extension conditions τ.

Definition 13 An MWCSP $\mathcal{P}$ is *DC-NC[τ]* (*DC-AC[τ]* resp.) iff $\mathcal{P}$ is DC-NC (DC-AC resp.), and all projection/extension conditions τ for both $\mathcal{P}$ and the normalized dual problem $\mathcal{P}^N$ are satisfied.

Definition 14 An MWCSP $\mathcal{P}$ is *DQ-NC[τ]* (*DQ-AC[τ]* resp.) iff $\mathcal{P}$ is DQ-NC (DQ-AC resp.), and all the projection/extension conditions τ for $\mathcal{P}$ are satisfied.

Previous work [21] shows experimental results on an implementation of DQ-NC[proj-NC*] and DQ-AC[proj-AC*], where DQ-NC[proj-NC*] and DQ-AC[proj-AC*] are named as node and arc consistency respectively.

5.5 Stronger Solution Definitions

This section discusses the scopes and limitations of our techniques on solving MWCSPs for the other two stronger solved levels: weakly solved and strongly solved.

In terms of space, the solution sizes for solving MWCSPs ultra-weakly, weakly, and strongly vary from $O(n)$, $O((n-m)d^m)$, to $O(d^n)$ respectively, where n is the total number of variables, $m \leq n$ is the number of variables owned by adversaries, and d is the maximum domain size of the MWCSP. A direct consequence is that we need exponential space to store weak/strong solutions during search, and most often, compact representations to represent weak/strong solutions are more desirable.

In terms of prunings in branch and bound tree search, a sound pruning condition when solving a weaker solution concept may not hold in stronger ones. This is

caused by the removal of the assumption of optimal/perfect plays when dealing with stronger solution concepts. For example in alpha-beta prunings, when the $\min$ player obtains an A-cost which is lower than the lb (i.e. $\max$ player's last found best), we cannot immediately backtrack if we want to tackle weakly solved solutions, where we assume the $\max$ player is the adversary. The reason behind is that we cannot assume the $\max$ player must play a perfect move. We have to consider all moves for the $\max$ player. The situation is similar if we assume the $\min$ player is the adversary. By similar reasoning and inductions, we cannot perform prunings/backtrackings for the $\leq lb$ column ($\geq ub$ column resp.) in Table 1 if we want to tackle weakly solved solutions, assuming the $\max$ player ($\min$ player resp.) is the adversary. For solving strong solutions, the situation is even worse. We cannot assume optimal plays for both players. Therefore, we have to find A-costs for all sub-problems, and all prunings/backtrackings conditions in Table 1 cannot be used. In general, the fewer sound pruning/backtracking conditions available, the larger the search space we have to search. By using tree search, we can observe finding stronger solutions is much harder than weaker ones.

When tackling real-life problems, one can ask for solutions which solve the problem in an intermediate level. For example, if the adversaries have multiple optimal strategies, we can require solutions containing responses to every different optimal choice the adversaries may choose. In this case, the solved level lies between ultra-weak and weak. One way to handle is to relax the bound updating procedure for the lower bound (upper bound resp.) in alpha-beta pruning (Line 8 and 10 in Figure 3), where we assume the $\max$ ($\min$ resp.) player is the adversary. When a larger lower bound lb (smaller upper bound ub resp.) is found, we update the lower bound to $lb-1$ (upper bound to $ub+1$ resp.). The major focus of this paper is to give consistency notions to improve the search in finding the best-worst case, i.e. ultra-weak solutions, of a game.

It is also worthwhile to note that ultra-weak solutions provide the value of the initial state and the optimal play of the first step. If we are allowed to re-compute ultra-weak solutions after every step/move from our adversary, we could still achieve an optimal strategy. In this case, we avoid building exponential weak-solutions by re-computing ultra-weak solutions multiple times (i.e. trading time for space).

6 Performance Evaluation

In this section, we compare our solver in seven modes: Alpha-beta pruning, DC-NC[proj-NC*], DQ-NC[proj-NC*], DC-AC[proj-AC*], DQ-AC[proj-AC*], DC-AC[proj-FDAC*], and DQ-AC[proj-FDAC*] [15]. Values are labeled in static lexicographic order. We generate 20 instances for each benchmark's particular parameter setting. We set $k = \infty$ in all of the benchmarks. To ease our implementation, ∞ will be translated to a large enough constant. Results for each benchmark are tabulated with the average time used (in sec.) and average number of tree nodes encountered. We take average for solved instances *only*. If there are any unsolved instances, we give the number of solved instances beside the average time (superscript in brackets). Winning entries are highlighted in bold. Note that even if an entry runs slower

or encounter more search nodes than another entry, the entry will still be winning if it solves more instances than the others. A symbol '-' represents all instances fail to run within the time limit. The experiment is conducted on a Core2 Duo 2.8GHz with 3.2GB memory. We have also performed experiments on QeCode, a solver for QCOPs [4], by transforming the instances to QCOPs according to the transformation in previous work [21].

6.1 Randomly Generated Problems

We re-use benchmark MWCSP instances by Lee, Mak, and Yip [21]. The random MWCSP instances are generated with parameters (n, d, p), where n is the number of variables, d is the domain size for each variable, and p is the probability for a binary constraint to occur between two variables. There are no unary constraints which makes the instances harder, and the costs for each binary constraint are generated uniformly in [0..30]. Quantifiers are generated randomly with half probability for min (max resp.), and the number of quantifier levels vary from instances to instances. Time limit for the benchmark is set to 900 seconds, and Table 2 shows the results.

6.2 Graph Coloring Games

We re-use benchmark graph coloring game instances by Lee, Mak, and Yip [21]. We represent colors by numbers and the graph is numbered by two players. We partition the nodes into two sets A and B. Player 1 (Player 2 resp.) will number set A (B resp.). The goal of player 1 is to maximize the total difference between numbers of adjacent nodes, while player 2 wishes to minimize. The aim is to help player 1 extracting the best-worst case. The instances are generated with parameters (v, c, d), where v is

Table 2 Randomly Generated Problem

	Alpha-beta		DC-NC[proj-NC*]		DC-AC[proj-AC*]		DC-AC[proj-FDAC*]	
(n, d, p)	Time	#nodes	Time	#nodes	Time	#nodes	Time	#nodes
(12, 5, 0.4)	68.20	5,967,461	5.89	131,468	2.54	30,165	**2.13**	**20,397**
(12, 5, 0.6)	52.05	4,782,541	4.63	101,690	2.61	26,093	**2.24**	**16,178**
(14, 5, 0.4)	263.04$^{(18)}$	19,770,953	52.72	948,783	19.33	198,476	**14.82**	**117,155**
(14, 5, 0.6)	271.72$^{(17)}$	17,249,858	70.12	1,185,087	29.97	246,459	**23.11**	**143,197**
(16, 5, 0.4)	517.24$^{(2)}$	26,269,025	332.65$^{(19)}$	4,617,612	121.78	1,047,900	**102.82**	**706,913**
(16, 5, 0.6)	693.31$^{(2)}$	36,315,673	461.68$^{(16)}$	6,157,070	259.51	1,816,642	**208.52**	**1,054,326**
(18, 5, 0.4)	-	-	624.15$^{(5)}$	5,850,276	424.34$^{(9)}$	1,874,750	369.48$^{(12)}$	1,158,340
(18, 5, 0.6)	-	-	-	-	555.48$^{(5)}$	1,890,490	**515.69**$^{(9)}$	**1,127,819**
	QeCode		DQ-NC[proj-NC*]		DQ-AC[proj-AC*]		DQ-AC[proj-FDAC*]	
(n, d, p)	Time	#nodes	Time	#nodes	Time	#nodes	Time	#nodes
(12, 5, 0.4)	-	–	3.68	158,179	3.23	53,845	4.27	58,619
(12, 5, 0.6)	-	-	2.85	118,401	3.24	41,596	4.17	45,698
(14, 5, 0.4)	-	-	33.39	1,135,378	26.20	369,185	41.74	482,053
(14, 5, 0.6)	-	-	46.81	1,510,946	45.85	450,407	68.63	522,715
(16, 5, 0.4)	-	-	217.13	5,780,075	141.07	1,654,538	173.96	1,745,527
(16, 5, 0.6)	-	-	364.51$^{(19)}$	9,401,844	341.71	3,071,036	362.12$^{(17)}$	2,659,294
(18, 5, 0.4)	-	-	381.96$^{(5)}$	7,007,289	**582.85**$^{(13)}$	**4,297,854**	466.30$^{(8)}$	2,576,363
(18, 5, 0.6)	-	-	810.99$^{(3)}$	14,922,549	544.37$^{(3)}$	3,271,773	389.58$^{(4)}$	1,042,342

Table 3 Graph Coloring Game

	Alpha-beta		DC-NC[proj-NC*]		DC-AC[proj-AC*]		DC-AC[proj-FDAC*]	
(v, c, d)	Time	#nodes	Time	#nodes	Time	#nodes	Time	#nodes
(14, 4, 0.4)	19.88	1,572,978	6.71	122,266	3.20	37,252	**1.90**	**16,732**
(14, 4, 0.6)	24.12	1,730,473	10.38	185,111	5.88	59,359	**3.48**	**23,515**
(16, 4, 0.4)	167.75	10,050,800	48.37	688,200	22.67	221,484	**12.09**	**92,875**
(16, 4, 0.6)	166.83	9,213,029	45.71	625,944	27.03	212,934	**15.64**	**85,920**
(18, 4, 0.4)	784.47$^{(3)}$	33,914,968	288.90	2,839,962	114.63	792,220	**65.58**	**357,457**
(18, 4, 0.6)	-	-	350.29	3,400,265	163.70	993,099	**80.06**	**343,146**
(20, 4, 0.4)	-	-	653.32$^{(8)}$	5,559,267	545.54$^{(12)}$	2,206,506	**413.91**	**1,168,221**
(20, 4, 0.6)	-	-	-	-	724.17$^{(4)}$	2,498,710	**532.10**$^{(19)}$	**1,133,238**
	QeCode		DQ-NC[proj-NC*]		DQ-AC[proj-AC*]		DQ-AC[proj-FDAC*]	
(v, c, d)	Time	#nodes	Time	#nodes	Time	#nodes	Time	#nodes
(14, 4, 0.4)	-	-	4.52	170,843	3.36	63,298	3.74	53,722
(14, 4, 0.6)	-	-	7.29	269,179	6.36	99,972	6.88	74,187
(16, 4, 0.4)	-	-	34.43	1,002,145	23.21	363,539	24.36	281,229
(16, 4, 0.6)	-	-	33.82	949,861	29.19	352,694	31.99	280,426
(18, 4, 0.4)	-	-	204.86	4,095,993	118.65	1,315,346	140.95	1,207,566
(18, 4, 0.6)	-	-	267.23	5,295,433	180.38	1,711,948	182.66	1,270,797
(20, 4, 0.4)	-	-	542.40$^{(10)}$	9,356,227	538.00$^{(13)}$	3,990,062	459.01$^{(12)}$	2,390,484
(20, 4, 0.6)	-	-	793.10$^{(4)}$	13,240,872	761.80$^{(5)}$	4,698,459	689.72$^{(4)}$	2,952,531

Table 4 Generalized Radio Link Frequency Assignment Problem

	Alpha-beta		DC-NC[proj-NC*]		DC-AC[proj-AC*]		DC-AC[proj-FDAC*]	
(i, n, d, r)	Time	#nodes	Time	#nodes	Time	#nodes	Time	#nodes
(0, 24, 4, 0.2)	-	-	52.98	275,929	17.32	32,088	14.46	**20,396**
(1, 24, 4, 0.2)	-	-	86.38	442,362	50.54	74,182	53.85	**55,988**
(0, 24, 4, 0.4)	-	-	148.87	828,286	105.95	295,743	128.01	**286,122**
(1, 24, 4, 0.4)	-	-	168.54	905,277	122.50	289,965	154.00	**277,569**
(1, 22, 6, 0.2)	-	-	618.93	3,580,885	**307.58**	352,439	309.63	**299,361**
(0, 24, 6, 0.2)	-	-	1230.33$^{(19)}$	6,822,412	500.18	738,245	479.50	**651,762**
	QeCode		DQ-NC[proj-NC*]		DQ-AC[proj-AC*]		DQ-AC[proj-FDAC*]	
(i, n, d, r)	Time	#nodes	Time	#nodes	Time	#nodes	Time	#nodes
(0, 24, 4, 0.2)	-	-	28.15	279,590	15.25	35,455	**11.77**	23,880
(1, 24, 4, 0.2)	-	-	**45.62**	449,164	50.75	77,286	47.08	62,734
(0, 24, 4, 0.4)	-	-	**96.55**	1,046,150	101.49	451,090	208.79	692,470
(1, 24, 4, 0.4)	-	-	115.26	1,205,458	**109.16**	348,040	224.62	576,335
(1, 22, 6, 0.2)	-	-	338.42	3,719,348	374.34	374,385	309.96	368,643
(0, 24, 6, 0.2)	-	-	682.60$^{(19)}$	7,224,677	539.69	803,087	**434.99**	812,048

an even number of nodes in the graph, c is the range of numbers allowed to place, and d is the probability of an edge between two vertices. Player 1 (Player 2 resp.) is assigned to play the odd (even resp.) numbered turns, and the node corresponding to each turn is generated randomly. Time limit is set to 900 seconds, and Table 3 shows the results.

6.3 Generalized Radio Link Frequency Assignment Problem (GRLFAP)

We generate the GRLFAP according to two small but hard CELAR sub-instances [8], which are extracted from CELAR6. All GRLFAP instances are generated with parameters (i, n, d, r), where i is the index of the CELAR sub-instances ($\texttt{CELAR6-SUB}_i$), n is an even number of links, d is an even number of allowed frequencies, and r is the ratio of links placed in unsecured areas, $0 \leq r \leq 1$. For each instance, we

randomly extract a sequence of n links from $\texttt{CELAR6-SUB}_i$ and fix a domain of d frequencies. We randomly choose $\lfloor (r \times n + 1)/2 \rfloor$ pairs of links to be unsecured. If two links are restricted not to take frequencies f_i and f_j with distance less than t, we measure the costs of interference by using a binary constraint with violation measure $\max(0, t - |f_i - f_j|)$. We set the time limit to 7200 seconds. Table 4 shows the results.

6.4 Results and Discussions

For all benchmarks, all six consistencies are significantly faster and stronger than alpha-beta pruning.

Comparing the two duality approaches, we observe that duality of constraints (DC) has a smaller search space than duality of quantifiers (DQ). We conjecture for any projection/extension conditions τ, DC-NC[τ] (DC-AC[τ] resp.) could be stronger than DQ-NC[τ] (DQ-AC[τ] resp.). Note that enforcing projection/extension conditions on DQ-NC/DQ-AC may strengthen one approximation function, and weaken the other at the same time. DC-NC/DC-AC extracts costs from different copies of constraints and resolves this issue.

For all benchmarks, DQ-NC[proj-NC*] runs faster than DC-NC[proj-NC*]. In almost all instances of randomly generated problems and the graph coloring game, DC-AC[proj-(FD)AC*] runs faster than DQ-AC[proj-(FD)AC*], with DC-AC[proj-FDAC] the fastest. A notable exception is (18, 5, 0.4) in randomly generated problems, where DQ-AC[proj-AC*] manages to solve more instances than DC-AC[proj-AC*] and even DC-AC[proj-FDAC*]. However, for instances (in that parameter setting) which could be solved by both DQ-AC[proj-AC*] and DC-AC[proj-FDAC*] (a total of 11 instances), we observe that DC-AC[proj-FDAC*] runs significantly faster. In GRLFAP, DQ-NC[proj-NC*] runs faster than the others for smaller instances (except (0, 24, 4, 0.2)) and stronger consistencies tend to be faster for larger ones. Enforcing proj-FDAC* is more computationally expensive than proj-AC* and proj-NC*, and implementing duality of constraints requires implementing two copies of constraints. Therefore, stronger consistencies are worthwhile for larger instances, but not for smaller ones due to the large computational over-head.

It is worth noting in some particular instances, DQ[proj-FDAC*] prunes less than DQ[proj-AC*]. This could be explained by the fact that adding stronger projection/extension conditions from Weighted CSPs naively does not always strengthen our approximation functions. We may have to further study and consider quantifier information.

All QCOP instances for even the smallest parameter settings for all benchmarks fail to run within the time limit. QCOPs are, in fact, more general [21] than MWCSPs. By viewing a more specific problem, it is natural for us to devise consistency techniques outperforming QeCode.

One interesting observation is that we may be able to combine the two duality approaches to form a even stronger level of consistencies. However, maintaining a stronger level of consistencies by naively combining the two enforcing algorithms could incur an expensive overhead on the propagation routine and increases the overall runtime. We have to balance the amount of time spending on search and propa-

gations. In this paper, we therefore only allow the solver to choose either duality of constraints or duality of quantifiers.

7 Concluding Remarks

Our contributions are five-fold. First, we formally define the Minimax Weighted CSP framework for modeling optimization problems with adversaries. Our work allows us to model and solve soft constraint problems with adversaries, such as the graph coloring game and the generalized radio link frequency assignment problem. Second, we implement a complete solver incorporating alpha-beta pruning into branch-and-bound, and propose sufficient pruning and backtracking conditions which serve as a basis for our consistency notions. Third, we define and implement node and (full directional) arc consistency notions to reduce the search space of an alpha-beta search for Minimax Weighted CSPs, by approximating lower and upper bounds of the cost of the problem. Lower bound computation employs standard estimation of costs in the sub-problems and we propose two approaches based on the Duality Principle to estimate upper bounds. Fourth, we show how to adopt and re-use Weighted CSP consistencies to strengthen our lower and upper approximation functions, and also discuss capabilities and limitations of our approach on other stronger solution concepts. Fifth, we perform experiments on comparing basic alpha-beta pruning and the six consistencies from the two dualities.

There are also two closely related frameworks, where both tackle constraint problems with adversaries. Brown et al. propose adversarial CSPs [7], which focuses on the case where two opponents take turns to assign variables, each trying to direct the solution towards their own objectives. Another related work is Stochastic CSPs [34], which can represent adversaries by known probability distributions. Their work focuses on seeking actions to minimize/maximize the expected cost for all the possible scenarios. Our work is similar in the sense that we are minimizing/maximizing costs for the worst case scenario.

Other possible future work includes: incorporating high arity soft table/global constraints similar to those for Weighted CSPs [18,19,22], value and variable ordering heuristics [20], theoretical comparisons on different consistency notions, and tackling stronger solutions. Devising online/distributed algorithms for Minimax Weighted CSPs is also an interesting future work.

References

1. Allis, L.V.: Searching for solutions in games and artificial intelligence. Ph.D. thesis, University of Limburg (1994)
2. Apt, K.: Principles of Constraint Programming. Cambridge University Press, New York, USA (2003)
3. Arora, S., Barak, B.: Computational Complexity: A Modern Approach, 1st edn. Cambridge University Press (2009)
4. Benedetti, M., Lallouet, A., Vautard, J.: Quantified constraint optimization. In: CP'08, pp. 463–477 (2008)
5. Bordeaux, L., Cadoli, M., Mancini, T.: CSP properties for quantified constraints: Definitions and complexity. In: AAAI'05, pp. 360–365 (2005)

6. Bordeaux, L., Monfroy, E.: Beyond NP: Arc-consistency for quantified constraints. In: CP'02, pp. 371–386 (2002)
7. Brown, K.N., Little, J., Creed, P.J., Freuder, E.C.: Adversarial constraint satisfaction by game-tree search. In: ECAI'04, pp. 151–155 (2004)
8. Cabon, B., de Givry, S., Lobjois, L., Schiex, T., Warners, J.: Radio link frequency assignment. CONSTRAINTS **4**, 79–89 (1999)
9. Cooper, M., de Givry, S., Sanchez, M., Schiex, T., Zytnicki, M., Werner, T.: Soft arc consistency revisited. Artificial Intelligence **174**(7-8), 449–478 (2010)
10. Cooper, M.C., De Givry, S., Schiex, T.: Optimal soft arc consistency. In: IJCAI'07, pp. 68–73 (2007)
11. Debruyne, R., Bessiere, C.: From restricted path consistency to max-restricted path consistency. In: CP'97, pp. 312–326 (1997)
12. Dempe, S.: Foundations of Bilevel Programming. Kluwer Academic Publishers (2002)
13. Gent, I.P., Nightingale, P., Stergiou, K.: QCSP-Solve: A solver for quantified constraint satisfaction problems. In: IJCAI'05, pp. 138–143 (2005)
14. van den Herik, H.J., Uiterwijk, J.W.H.M., van Rijswijck, J.: Games solved: Now and in the future. Artif. Intell. **134**(1-2), 277–311 (2002)
15. Lallouet, A., Lee, J.H.M., Mak, T.W.K.: Consistencies for ultra-weak solutions in minimax weighted csps using the duality principle. In: CP'12, pp. 373–389 (2012)
16. Larrosa, J., Schiex, T.: In the quest of the best form of local consistency for weighted CSP. In: IJCAI'03, pp. 239–244 (2003)
17. Larrosa, J., Schiex, T.: Solving weighted CSP by maintaining arc consistency. Artificial Intelligence **159**(1-2), 1–26 (2004)
18. Lee, J.H.M., Leung, K.L.: Consistency techniques for flow-based projection-safe global cost functions in weighted constraint satisfaction. JAIR **43**, 257–292 (2012)
19. Lee, J.H.M., Leung, K.L., Wu, Y.: Polynomially decomposable global cost functions in weighted constraint satisfaction. In: AAAI'12, pp. 507–513 (2012)
20. Lee, J.H.M., Mak, T.W.K.: A value ordering heuristic for solving ultra-weak solutions in minimax weighted csps. In: ICTAI'12, pp. 17–24 (2012)
21. Lee, J.H.M., Mak, T.W.K., Yip, J.: Weighted constraint satisfaction problems with min-max quantifiers. In: ICTAI '11, pp. 769–776 (2011)
22. Lee, J.H.M., Shum, Y.W.: Modeling soft global constraints as linear programs in weighted constraint satisfaction. In: ICTAI '11, pp. 305–312 (2011)
23. Mamoulis, N., Stergiou, K.: Algorithms for quantified constraint satisfaction problems. In: CP'04, pp. 752–756 (2004)
24. Murty, K.G.: Linear and Combinatorial Programming. R. E. Krieger (1985)
25. Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V.V.: Algorithmic Game Theory. Cambridge University Press (2007)
26. Pralet, C., Schiex, T., Verfaillie, G.: Decomposition of multi-operator queries on semiring-based graphical models. In: CP'06, pp. 437–452 (2006)
27. Pralet, C., Schiex, T., Verfaillie, G.: Sequential Decision-Making Problems - Representation and Solution. Wiley (2009)
28. Prieditis, A.E., Fletcher, E.: Two-agent ida*. Journal of Experimental & Theoretical Artificial Intelligence **10**(4), 451–485 (1998)
29. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA (2006)
30. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Pearson Education (2003)
31. Schaeffer, J., Burch, N., Bjrnsson, Y., Kishimoto, A., Mller, M., Lake, R., Lu, P., Sutphen, S.: Checkers is solved. Science **317**(5844), 1518–1522 (2007)
32. Sturtevant, N.R., Korf, R.E.: On pruning techniques for multi-player games. In: AAAI'00, pp. 201–207 (2000)
33. Von Neumann, J., Morgenstern, O.: Theory of Games and Economic Behavior. Princeton University Press (1944)
34. Walsh, T.: Stochastic constraint programming. In: ECAI '02, pp. 111–115 (2002)
35. Wolsey, L.A.: Integer Programming. Wiley (1998)
36. Wu, Y.: Tractable projection-safe soft global constraints in weighted constraint satisfaction. Master's thesis, The Chinese University of Hong Kong (2011)