master's thesis

Implementation and Extension of the Virtual Arc Consistency Algorithm for Weighted Constraint Satisfaction Problem

Lucie Bužková

June 2015

Ing. Tomáš Werner, Ph.D.

Czech Technical University in Prague

Faculty of Electrical Engineering, Department of Cybernetics

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	Bc. Lucie B u ž k o v á
Studijní program:	Otevřená informatika (magisterský)
Obor:	Počítačové vidění a digitální obraz
Název tématu:	Implementace a rozšíření algoritmu na virtuální hranovou konzistenci pro problém váženého programování s omezujícími podmínkami

Pokyny pro vypracování:

1. Implementujte algoritmus pro uvedení problému váženého CSP (weighted constraint satisfaction problém, WCSP) do stavu virtuální hranové konzistence (virtual arc consistency, VAC) podle [1].
2. Algoritmus rozšiřte tak, aby WCSP uvedl do stavu virtuální J-konzistence definované v [2].
3. Algoritmus rozšiřte tak, aby (alespoň částečně) využíval informace z předešlé iterace a byl tak použitelný na velké instance (např. obrazy). Inspirujte se předešlou implementací příbuzného algoritmu [3,4].
4. Zamyslet se nad tím, jak by algoritmus šlo rozšířit, aby umožňoval co nejefektivnější 'warm-start' po malé změně instance WCSP (např. přidání/odebrání/změna jediného omezení) či úrovně J-konzistence. Tato vlastnost je klíčová, má-li se algoritmus použít ve smyčce branch&bound či branch&cut.

Implementace bude v Matlabu (možno ale zvolit i jiný jazyk, např. C++ nebo Java). Kromě výsledného kódu bude výstupem pseudokód napsaný v terminologii a značení používaných v [2-5].
Práce má výzkumný charakter. Jejím hlavním přínosem má být:
1. Co nejjednodušší a nejpromyšlenější algoritmus.
2. Pečlivé odůvodnění a otestování správnosti algoritmu.
3. Diskuze překážek, které se po cestě vyskytnou, a možných cestách k jejich řešení.

Získané zkušenosti by měly být krokem k budoucí rychlé (C++) implementaci všestranně použitelného algoritmu umožňujícího velmi efektivní ořezávání hledacího prostoru WCSP.

Seznam odborné literatury:

[1] MC Cooper, S de Givry, M Sanchez, T Schiex, M Zytnicki, T Werner: Soft Arc Consistency Revisited. Artificial Intelligence 174(7-8):449-478, May 2010.
[2] T Werner: Revisiting the Linear Programming Relaxation Approach to Gibbs Energy Minimization and Weighted Constraint Satisfaction. IEEE Trans. on Pattern Recognition and Machine Intelligence (PAMI) 32(8) August 2010.
[3] T Werner: A Linear Programming Approach to Max-sum Problem: A Review. IEEE Trans. on Pattern Recognition and Machine Intelligence (PAMI) 29(7), July 2007.
[4] T Werner: A Linear Programming Approach to Max-sum Problem: A Review. Research report CTU-CMP-2005-25, Dec 2005.
[5] V Franc, S Sonnenburg, T Werner: Cutting Plane Methods in Machine Learning. A chapter in: Optimization for Machine Learning, MIT Press, 2012.

Vedoucí diplomové práce: Ing. Tomáš Werner, Ph.D.

Platnost zadání: do konce letního semestru 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic	prof. Ing. Pavel Ripka, CSc.
vedoucí katedry	**děkan**

V Praze dne 17. 2. 2015

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student:	Bc. Lucie B u ž k o v á
Study programme:	Open Informatics
Specialisation:	Computer Vision and Image Processing
Title of Diploma Thesis:	Implementation and Extension of the Virtual Arc Consistency Algorithm for Weighted Constraint Satisfaction Problem

Guidelines:

1. Implement the algorithm to enforce VAC (virtual arc consistency) for weighted constraint satisfaction problem (WCSP) according to [1].
2. Extend the algorithm to enforce J-consistency, as defined in [2].
3. Extend the algorithm to (at least partially) use information from previous iterations, which makes it applicable to large instances (such as images). Take ideas from the existing implementation of the related algorithm [3-4].
4. Consider an extension of the algorithm to support 'warm-start' after a small change of the WCSP instance (such as adding/deleting/modification of a single constraint) or the level of J-consistency. This property is crucial for using the algorithm withing a branch&bound or branch&cut search loop.

The implementation will be done in Matlab (but other languages are possible too, such as C++ or Java). Besides the resulting code, an outcome of the work will be a pseudocode, written in the notation and terminology of [2-5].
The work has a research flavour. Its main expected outcome is:
1. The algoritm, as simple as possible and thoroughly thought over.
2. Thorough justification and testing of the correctness of the algorithm.
3. A discussion of the difficulties encountered and of possible ways of their solution.

The gained experiences should be a step towards a future fast (C++) implementation of a versatile algorithm to very effectively prune the WCSP search space.

Bibliography/Sources:

[1] MC Cooper, S de Givry, M Sanchez, T Schiex, M Zytnicki, T Werner: Soft Arc Consistency Revisited. Artificial Intelligence 174(7-8):449-478, May 2010.
[2] T Werner: Revisiting the Linear Programming Relaxation Approach to Gibbs Energy Minimization and Weighted Constraint Satisfaction. IEEE Trans. on Pattern Recognition and Machine Intelligence (PAMI) 32(8) August 2010.
[3] T Werner: A Linear Programming Approach to Max-sum Problem: A Review. IEEE Trans. on Pattern Recognition and Machine Intelligence (PAMI) 29(7), July 2007.
[4] T Werner: A Linear Programming Approach to Max-sum Problem: A Review. Research report CTU-CMP-2005-25, Dec 2005.
[5] V Franc, S Sonnenburg, T Werner: Cutting Plane Methods in Machine Learning. A chapter in: Optimization for Machine Learning, MIT Press, 2012.

Diploma Thesis Supervisor: Ing. Tomáš Werner, Ph.D.

Valid until: the end of the summer semester of academic year 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic	prof. Ing. Pavel Ripka, CSc.
Head of Department	**Dean**

Prague, February 17, 2015

Acknowledgement

Ráda bych tímto poděkovala své rodině za velkou podporu během studií. Velké poděkovaní také patří mému vedoucímu práce za konzultace a přínosné rady, které vedly k sepsání této diplomové práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracovala samostatně, a že jsem uvedla veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V praze dne 10.5.2015 ....

Lucie Bužková

Abstract

Problém splňování vážených podmínek (WCSP) je optimalizační problém daný množinou proměnných a množinou funkcí definovaných nad nimi. Mnoho problémů z oblasti zpracování obrazu, rozvrhování nebo bioinformatiky bylo formulováno jako WCSP. WCSP patří do třídy NP-úplných úloh, snahou je však nalézt algoritmy, které prořezávají vyhledávací prostor. Algoritmus VAC je založen na postupném zvyšování dolní meze řešení pomocí propagování virtuální hranové konzistence.

V této diplomové práci rozšíříme algoritmus VAC, aby propagoval virtuální J-konzistenci, která kontroluje konzistenci pouze u podmínek zahrutých v množině J. Tenhle přístup umožňuje kompromis mezi rychlostí a kvalitou spodní meze. Dále také vylepšíme algoritmus VAC tím, že budeme využívat informace z předchozích iterací, čímž zabráníme redundantním testům konzistence. Obě implementace jsou porovnány a vyhodnoceny na několika datasetech.

Klíčová slova

Programování s omezujcími podmínkami, Problém splňování vážených podmínek, Konzistence pro WCSP

Abstract

Weighted constraint satisfaction problem (WCSP) is an optimization problem given by a set of cost functions defined over discrete variables. Many problems in areas such as image processing, scheduling or bioinformatics have been formulated as WCSP. WCSP belongs to the class of NP-complete problems, however, there has been an effort to develop algorithms to prune the search space. VAC algorithm is based on incremental increase in the lower bound on any solutions by enforcing the virtual arc consistency.

In this thesis, we extend the VAC algorithm to enforce the J-consistency that proves consistent only constraints involved in a set J. This enable a trade-off between time and the quality of the solution. Furthermore, we improve the VAC algorithm by storing some information from the previous iteration to be used later. Both implementation are compared and evaluated on several datasets.

Keywords

Constraint programming, Weighted constraint satisfaction problem; Soft constraint; Soft consistency

Contents

1 INTRODUCTION

In last years, constraint programming (CP) has been successfully applied in many areas, such as scheduling, networks and computer vision. CP technigues have also become a part of engine in many solvers for discrete optimization [1]. The general idea of CP is to offer a powerful tool for solving the problems that can be naturally described by means of constraints.

Constraint programming covers two large areas. On one hand, it deals with the formulation and representation of problems. The main class of problems solved by CP is called constraint satisfaction problems (CSP). A problem is defined in a natural language of decision variables and constraints and then a solver is used to find a solution. On the other hand, the CP includes a huge amount of algorithms and techniques to solve the problems.

Since CSP belongs to the class of NP-complete problems, we cannot expect an outstanding performance for all the problems of any size. Any algorithm running in polynomial time is not known to solve the problems in NP-complete class. However, we can focus on particular properties of the problems and solve them more efficiently.

Chapter 2 is devoted to the study of the general framework of constraint programming. We set up notation and terminology concerning the constraint networks used in this thesis. Variables and constraints are presented here. Moreover, we touch the concept of constraint support which is necessary for the definition of consistency that is studied in chapter 3. First-order consistencies are described in the first part of the chapter. We proceed with the study of higher order consistencies identifying consistency of several values simultaneously.

Some real problems are over-constrained and have no feasible solution. However, a penalty can be paid for the constraint violation. Hence, a new problem rises: finding a solution with minimal penalty. Here, a weighted constraint satisfaction problem (WCSP) is introduced. Adding weights to constraints gives us freedom to express preferences among solutions. Chapter 4 provides an itroduction into the concept of WCSP, which includes the operations on soft constraints and the consistency of soft constraints.

A WCSP is one of possible extensions of CSP. A classical CSP is a set of variables and hard (crisp) constraints specifying tuples that are allowed. WCSP replaces crisp constraints by soft constraints defined by a cost function. The total cost can be limited by both upper and lower bound that can be used e.g. in branch and bound algorithm. Bounds on WCSP have been studied in [1] or

[1]Solvers using CP: ILOG CP Optimizer OPL, SICStus Prolog, Eclipse CLP, JaCoP, MINION etc.

[2].

VAC algorithm for increasing the lower bound on solution of WCSP was introduced in [3]. The algorithm enforces soft arc consistency by applying a planned sequence of soft arc consistency operations which necessarily leads to increase in lower bound.

In this thesis, we extend this algorithm to enforce J-consistency based on pairwise consistency. Section 5 provides a detailed exposition of the extended algorihtm. Unlike pairwise consistency enforcing consistency between all pairs of constraints, J-constancy proves consistent only pairs listed in a set J.

Assuming that unary constraints coincide with domains of variables, general arc consistency can be transformed into J-consistency. J-consistency can be both weaker and stronger than general arc consistency depending on the number of pairs in J. This enables us a trade-off between time and solution.

Furthermore, in chapter 6, we introduce an extension of the VAC algorithm so that it remembers information reached in the previous iteration. Finally, the results are summed up in chapter 7.

2 GENERAL CONCEPT OF CONSTRAINT PROGRAMMING

Many problems consist in finding an assignment of values to a number of variables under certain conditions, which leads to the creation of constraint programming concept. This concept includes two important components: variables and constraints. In this chapter, we describe these two objects and their properties. Then we formulate the constraints satisfaction problem.

2.1 Variables

Definition 2.1 (Variable and domain) *In general, variable is an object having a name and taking on a value from a set of values, called a domain. We write X_a for the domain of variable a and $x_a \in X_a$ for the fact that a value x_a belongs to the domain of variable a.*

Notation 2.1.1 *For abbreviation, we will write a set of variables rather then a totally ordered set of variables when no confusion can arise. We emphases it by () notation.*

In constraint processing, the domain is always finite. A domain can change over time by ruling out some values, nevertheless a *current domain* is a subset of an *initial domain*. In case that a variable is *explicitly* assigned a value, every other value can be deleted from its domain. Values may be also removed implicitly by reasoning. A variable whose domain contains only one value is said to be *fixed*, and *unfixed* otherwise.

A value x_a is called *valid* iff it is contained in the domain of a. However, the value can be invalidated because of lack support (see Def. 2.11) on some constraint, hence a killer structure is introduced to record it. The fact that value x_a has been removed from X_a because of lack support on c is denoted by $killer_{x_a} = c$.

Example 2.2 *Given two variables a and b with domain $\{1, 2\}$ and an equality $a = b$ between them, variable a is assigned the value 1. Now we can see that b can take only the value 1 and the value 2 can be ruled out from the domain X_b. Hence, both variables are fixed, the first one explicitly and the other one implicitly. Denoting the equality $a = b$ by c, we will write $killer_{x_b} = c$.*

Analogously, we may assign a tuple of values to a set of variables. Here, we define a joint domain and a joint state as follows.

Definition 2.3 (Joint Domain, Joint State) *For a set of variables A, a joint domain, denoted by X_A, is said to be a Cartesian product of initial domains of variables in A, i.e. $X_A = \times_{a \in A} X_a$. An element $x_a \in X_A$ is a join state.*

Notation 2.3.1 *For convenience, if a set of variables $A = \{a\}$ contains only one variable, we write x_a instead of $x_{(a)}$.*

Example 2.4 *Let A = 1,2,3, $X_1 = X_2 = X_2 = \{a, b\}$. Then the joint state ordered lexicographically is*

$$X_A = \begin{Bmatrix} (a,a,a), \\ (a,a,b), \\ (a,b,a), \\ (a,b,b), \\ (b,a,a), \\ (b,a,b), \\ (b,b,a), \\ (b,b,b) \end{Bmatrix}$$

.

Definition 2.5 (Relation) *Given a set of variables A and a sequence of domains $X_a, a \in A$, a relation R_A over a set of variables A is, by definition, a subset of the Cartesian product $\prod_{a \in A} X_a$, hence $R_A \subseteq \prod_{a \in A} X_a$.*

Two relations that differ only in order of variables are considered to be same. Now let's discuss operations on relations applied in constraint programming. We take into consideration only two of them: projection and join.

Definition 2.6 (Projection) *Given two sets of variables $A \subseteq B$ and a relation defined over A, a projection of x_A on B, denoted by $\pi_B(R_A)$, is a set of all tuples in R_A restricted to values corresponding to variables in B, i.e. the scope of $\pi_B(R_A)$ is B.*

Definition 2.7 (Join) *Given two relations R_A and R_B, a natural join, denoted by $R_A \bowtie R_B$, is a set of tuples that are combined from tuples in R_A and R_B such that values corresponding to their common variables equal, i.e. the scope of $R_A \bowtie R_B$ is $A \cup B$. Left join $R_A \bowtie R_B$, right join $R_A \bowtie R_B$ respectively, is defined by $\pi_A(R_A \ltimes R_B)$, $\pi_B(R_A \rtimes R_B)$ respectively.*

Example 2.8 *We have two sets of variables $A = \{v_1, v_2, v_3\}$, $B = \{v_2, v_3, v_4\}$ and relations R_A, R_B defined over them given in Figure 1.*

R_A

v_1	v_2	v_3
1	1	2
2	1	2
2	2	2

R_B

v_2	v_3	v_4
1	2	3
1	3	3
2	2	3

$R_A \bowtie R_B$

v_1	v_2	v_3	v_4
1	1	2	3
2	1	2	3
2	3	2	3

$\pi_{v_2,v_3}(R_A \bowtie R_B)$

v_2	v_3
1	2
3	2

Figure 1 Example of operations join and projection

Relations are necessary to define constraints, an important component of constraint programming that restricts values in domains.

2.2 Constraints

Definition 2.9 (Constraint) *A constraint is an object defined by a relation over a set of variables containing exactly the set of tuples accepted by the constraint. A set of variables involved in a constraint is said to be a scope. A scope can be assigned a tuple of values called joint state. Hence, a constraint is a pair consisting of a scope and a relation, denoted by $(A, R_A), R_A \subseteq X_A$. The size of a scope is called an arity of the constraint.*

A tuple x_A accepted by a constraint c_A is also said to be allowed by c. We can also say that x_A satisfies c. On the other hand, a tuple x_A that is forbidden by c is said to violate or unsatisfy c.

A constraint $c = (A, R_A)$ is extensional iff the relation R_A is described by listing the tuples allowed, resp. disallowed, by c. An extensional constraints (A, R_A) can be represented by a list of tuples.

Since the size of domains changes over time, a joint state can become invalid. A valid tuple for a constrains is such a tuple whose values are valid for every variable in the scope, hence all values belong to the current domains of corresponding variables in the scope.

Definition 2.10 (Valid tuple) *Let c_A be a n-constraint over the scope A and $x_A = (x_{a_1}, x_{a_2}, \dots x_{a_n})$ be an n-tuple. The tuple x_A is said to be valid iff $\forall a \in A, x_a \in X_a$. We define a function $\bar{f}_A : X_A \rightarrow \{1, 0\}$ where $\bar{f}_{AX_A}(x_A) = 1$ when x_A is valid and 0 otherwise.*

Informally, a valid tuple belongs to the Cartesian product of all current domains in the scope and a set of valid tuples is a subset of joint states of the scope. Next, we define supports and conflicts.

Definition 2.11 (Support and conflict) *Given a constraint c_A, we call support (resp. a conflict) a tuple x_A that is both valid on c_A and allowed (resp. disallowed) by c_A.*

If a support x_A on constraint c contains a value x_a, formally $\pi_a(x_A) = x_a$, we say that x_A is a support for x_a on c.

Example 2.12 *Given two variables* $1, 2$ *and their domains* $X_1 = X_2 = \{(a, b)\}$ *and a constraint* $c = ((1, 2), \{(a, b)\})$.

- *A tuple* (a, a) *is a valid tuple, however, it is not allowed by c, hence it is a conflict.*
- *A tuple* (a, b) *is both valid and allowed by c, hence it is a support.*

Now, we delete value a_1*, hence* $X_1 = \{a\}$.

- *A tuple* (a, a) *is neither valid or allowed by c.*
- *A tuple* (a, b) *is not a valid tuple, however, it is allowed by c.*

2.3 Constraint Satisfaction Problem

After describing variables and constraints, we formulate a constraint satisfaction problem as follows.

Definition 2.13 (Constraint Satisfaction Problem (CSP)) *A CSP (also called constraint network) is defined by a triple* (V, X, C) *where* V *is a totally ordered set of* n *variables* $V = (v_1, v_2, \ldots, v_n)$*,* X *is a set of corresponding domains* $X = (X_{v_1}, X_{v_2}, \ldots, X_{v_n})$ *such that each variable* $v_i \in V$ *takes a value from its domain* X_{v_i}*,* C *is a set of constraints* $C = (C_1, C_2, \ldots, C_k)$*. Each constraint* C_i *is defined by a pair* $C_i = (A, X_A)$ *such that* $A \subseteq V$ *and* $X_A \subseteq D^A$.

It happens that a CSP includes two or more constraints with the same scope. Such a problem is not normalized.

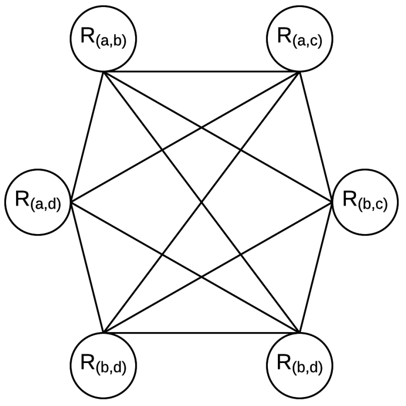
Definition 2.14 (Normalized Constraint Satisfaction Problem) *A CSP, defined by* (V, X, C)*, is said to be normalized iff* $\forall (A, R_A), (B, R_B) \in C$, $(A, R_A) \neq (B, R_B) \Rightarrow A \neq B$.

Every non normalized CSP can be transformed into a normalized CSP by replacing constraints (A, R_A) and (B, R_B) where $A = B$ with a new constraint defined by a tuple $(C, R_C), C = A = B, R_C = R_A \cap R_B$. In this thesis, we assume that every CSP is normalized.

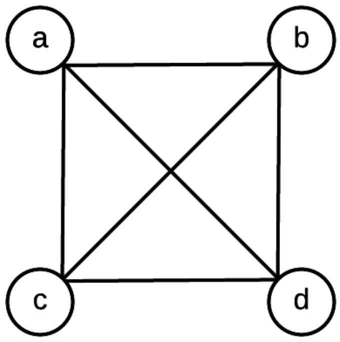
A constraint network can be represented by a (hyper)graph where variables stand for nodes and constraints stand for (hyper)edges. This hypergraph is said to be a constrained hypergraph.

There are two approaches how to represent a graph of a constraint network. The primal graph is based on variables corresponding to nodes. The constrained hypergraph is a primal graph. The other approach considers constraints to be nodes. Two nodes are connected by an edge iff the constraints corresponding to the nodes contain at least one variable. This graph is called a dual graph.

The second approach enables reformulate the whole CSP. Each constraint in the primal problem represents a variable in the dual problem. A domain in the dual problem consists of tuples allowed by the constraint in the original problem. New constraints are developed to enable constraint propagation, e.g. for each pair of variables (i.e. constraints in the primal problem) we create a new constraint whenever there is a non empty intersection of their scopes.



a) Primal graph



b) Dual graph

Figure 2 Primal and dual graph associated with the 4 queens problem in example 2.15

While the consistency techniques based on the primal problem remove values from domains of variables, the techniques based on the dual problem reduces number of valid tuples in constraints which are incompatible.

Example 2.15 *A 4 queens problem is a classical problem in computer science. It consists of finding a placement of four queens such that they do not threaten each other. Generally, there are $\binom{16}{4}$ possibilities how to place the queens. We can assume that each queen is placed in one columns, hence the task is to assign a position of the i^{th} queen within the i^{th} column. The problem is restricted to 4^4 possibilities.*

The four queens problem can be formulated as CPS such that each variable represents the position within a column and can take a value from $1 \dots 4$, i.e. number of the row (Fig 3a). Let us denote the queens by a, b, c and d respectively. Formally, let $V = (a, b, c, d)$ be a set of variables with domains $X_a = X_b = X_c = X_d = \{1, 2, 3, 4\}$ and C be a set of constraints defined relations as follows.

- $R_{(a,b)}$={(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}
- $R_{(a,c)}$={(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),(4,1),(4,3)}
- $R_{(a,d)}$={(1,2),(1,3),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,2),(4,3)}
- $R_{(b,c)}$={(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}
- $R_{(b,d)}$={(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),(4,1),(4,3)}
- $R_{(c,d)}$={(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}

The purpose of constraint programming is to find such an assignment to variables that meets all requirements. Here, we need a concept of instantiation and its extension. The basic idea is that an incompatible part of solution cannot lead to a correct solution.

Definition 2.16 (Instantiation) *Let $P = (V, X, C)$ be a constraint network.*

- *An instantiation I of a set of variables $A = \{a_1, a_2, \dots a_k\}$ is an assignment of values $x_{a_1}, x_{a_2}, \dots, x_{a_k}$ to variables in A, that is I is a sequence of tuples denoted by $\{(a_1, x_{a_1}), (a_2, x_{a_2}), \dots, (a_k, x_{a_k})\}$.*
- *An instantiation I on A is valid iff $\forall(a_i, x_{a_i}) \in I, \bar{f_{a_i}}(x_{a_i}) = 1$.*
- *An instantiation I on A is locally consistent iff it is valid and $\forall(B, R_B) \in C, b \in B$ such that $B \subseteq A$.*

Notation 2.16.1 *To emphasis that a variable a is assigned a value x, we write (a, x). Notation x_a refers to a value x that belongs to domain of a.*

Definition 2.17 (Extension) *Let I_1 be an instantiation on A, I_2 be an instantiation on B and $A \cap B = \emptyset$. An instantiation I on $A \cup B$ defined by $I = I_1 \cap I_2$ is said to be an extension of I_1 over B, or similarly, an extension of I_2 over A.*

Informally, an instantiation I on A is valid iff each value assigned to a variable in A belongs to its current domain and all constraints defined over a subset of A are satisfied. Given an instantiation, we can extend it by adding new variables and examine consistency of the new instantiation. The instantiation can be locally (2.16) or globally consistent. Global consistency of instantiation is defined as follows.

Definition 2.18 (Globally Consistent Instantiation) *An instantiation I on P is globally consistent iff it can be extended to a solution of P. It is globally inconsistent otherwise. The globally inconsistent instantiation is also called a nogood.*

It is evident that an instantiation which is locally inconsistent is inevitably globally inconsistent. However, a local consistency does not necessarily lead to the global consistency. An instantiation that is complete (i.e. covers the whole set of variables) and locally consistent is called a solution.

Definition 2.19 (Solution) *Given a network P = (V,X,C) a solution of P is an instantiation I on V that is locally consistent. Let $P' = (V, X', C')$ be a constraint network. Denoting the set of solution of P (P') by S (S' respectively), we say that P and P' are equivalent iff $S = S'$.*

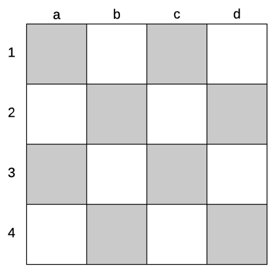
In the other words, two problems with the same set of variables are equivalent if the set of all solutions is same.

Example 2.20 *Consider the 4 queens problem (example 2.15). Let the first queen be placed in the first row, hence instantiation I_1 of a is $\{(a, 1)\}$.*

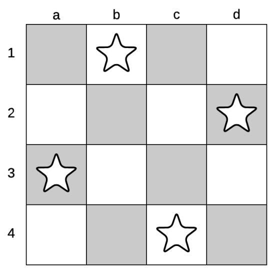
- *The instantiation $\{(a, 1)\}$ is locally consistent, since there is no constraint covered.*

Now, we extend I_1 over the variable b.

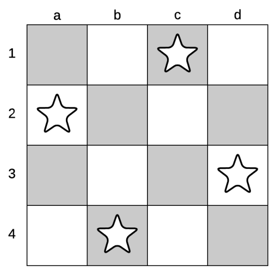
- *The instantiation $\{(a, 1), (b, 1)\}$ is valid, however, is locally inconsistent, since $(1, 1) \notin R_{(a,b)}$*



a) 4 queens problem



b) Solution of 4 queens problem



c) Solution of 4 queens problem

Figure 3 The 4 queens problem

- *The instantiation* $\{(a,1),(b,3)\}$ *is both valid and locally consistent, since* $(1,3) \in R_{(a,b)}$. $\{(a,1),(b,3)\}$ *is locally consistent extension of* $(a,1)$.

Let us $\{(a,1),(b,4)\}$ *denote by* I_2. *We extend* I_2 *over the variable c.*

- *The instantiation* $\{(a,1),(b,4),(c,2)\}$ *is both valid and locally consistent. However, it is not globally consistent because there is no value* $x_d \in X_d$ *such that instantiation* $\{(a,1),(b,4),(c,2),(d,x_d)\}$ *is locally consistent.*

The instantiation $\{(a,2),(b,4),(c,1),(d,3)\}$ *is locally consistent and since* $|I| = |V|$ *it is also a solution (Fig. 3c)*

A solution is an assignment to all variables satisfying all constraints. If a problem has a solution, it is called satisfiable and unsatisfiable otherwise.

In constraint programming, the basic task is to determine whether a problem is satisfiable or not. Besides that we can search for a solution that is optimal according to a given cost function. This is described in the chapter 4.

Since CSP is NP-complete problem, any algorithm running in polynomial time exists. However, there are many techniques pruning the search space and eliminating the size of the problem. Most of them are based on consistencies and their propagation.

3 CONSISTENCY

A consistency is a property of a constraint network used in many CP algorithms. A global consistency ensures that there exists a solution, however, it can be very expensive to find it. On the other hand, a local consistency involves only a subset of variables and/or constraints. A local consistency can help us to prune the search space and exclude values that cannot lead to any solution.

Example 3.1 *Let us consider an example (see Fig. 4). We have a set of three variables* $V = (a, b, c), X_a = X_b = X_c = \{1, 2, 3\}$ *and two relations defined over them:*

1. $a > b$,
2. $b + c = 5$.

There are 27 *possible combinations combinations of domains. If we take the first relation into consideration, it it obvious that the variable* a *can never take the value* 1*, hence instantiation* $(a, 1)$ *cannot lead to a solution and* 1 *is deleted from* X_a*. The same goes for value* 3 *in* X_b*. Observing the second relation, we can remove* 1 *from both* X_b *and* X_c*. Moreover, we remove value* 2 *from* X_c*, since it has no more a support in the relation. Now the second constraint is consistent. Since the domain of variable* b *has been changed, the consistency of the relation* $a < b$ *can be corrupted and we have to check it again. Now we can see that the value* $2 \in X_a$ *has lost its support, hence can be deleted. The domain of* b *has not been influence, hence the second constraint remains consistent. Now we can see that there is only one assignment:* $\{(a, 3), (b, 2), (c, 3)\}$.

This process, based on constraint propagation, is used by many filtering algorithms that may simplify the problem without loss of any solution. Before

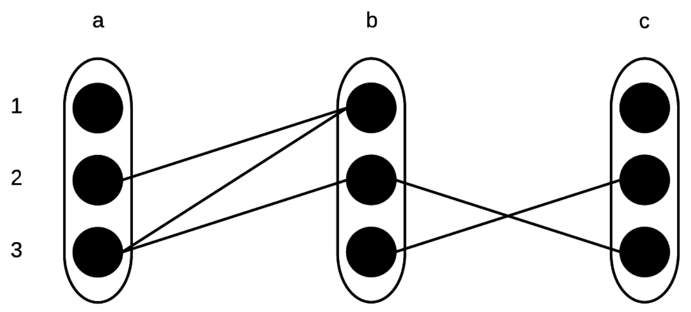


Figure 4 Graphical visualisation of constraints $a > b$ and $b + c = 5$

introducing filtering algorithms, we go through the most known consistencies. We distinguish two main classes of consistencies: domain based and constraint based consistencies. Domain based consistencies, which are also called first-order or domain-filtering consistencies, identify values in domains that are inconsistent. On the other hand, constraint based consistencies make constraints compatible. Neither of them modifies the constraint network hence any variable or constraint is not added.

Definition 3.2 (kth-order Consistency) *Let k be an integer $1 \leq k$. The kth-order consistency is a consistency detecting nogoods of size k.*

3.1 First-Order Consistency

First-order consistency identifies noogoods of size 1. It enables to remove inconsistent values from domains, however, it does not modify the constraint network. Node consistency and arc consistency belong to the most known consistencies and are defined as follows.

Definition 3.3 (Node Consistency, NC) *Let P = (V,X,C) be a CSP. The unary constraint c over variable a is node consistent iff every value in X_a satisfies c. P is node consistent iff every unary constraint is satisfied.*

Example 3.4 *Given a variable a with domain $X_a = \{1, 2, 3, 4\}$ and a constraint over a allowing only even number. The constraint is not node consistent since X_a contains values 1 and 3 that do not satisfy it. After X_A is restricted to $\{2, 4\}$, the constraint becomes node consistent.*

Unary constraints are closely related to the domain of variables. A problem is node consistent iff unary constraints coincide with unary constraints. It is useful to introduce a consistency covering more variables.

Definition 3.5 (General Arc Consistency, GAC) *Let P = (V,X,C) be a CSP.*

- *A constraint $c = (A, R_A) \in C$ is arc consistent relative to $a \in A$ iff $\forall x_a \in X_a$ there exists a support for x_A on c, hence $x_A \in R_A \ \wedge \ x_A \in X_A$.*
- *A constraint $c = (A, R_A) \in C$ is general arc consistent iff it is general arc consistent relative to every $a \in A$*
- *P is general arc consistent iff ti is general arc consistent for all variables in v on all constraints in C.*
- *Arc consistency (AC) is the general arc consistency for binary constraints (of arity 2).*

Note that the node consistency is the general arc consistency for unary constraints. For binary constraints, the general arc consistency is called arc consistency. GAC is one of the most important properties in constraint programming. It can be defined for constraint of arbitrary arity.

Effective algorithms to solve the arc consistency have always been the central point of constraint programming. Not only are they involved in most CP solvers, but also many ideas have come from in. Since the algorithm in this thesis is based on an AC algorithm, we will discuss it more in detail.

The most famous algorithm for consistency is known under the name AC3 [4]. It proves all constraints consistent by searching for a support. Whenever a domain of a variable is modified, all constraints over the variable have to be proved again. The process of searching for a support is called revision.

The Boolean revise function gets a variable and a constraint defined over it as an input and iterates the valid values in the domain. If the valid value cannot be extended into a locally consistent instantiation, it is removed from the domain. Whenever revise function modifies the domain it returns true and false otherwise.

For binary networks, the time complexity of AC3 algorithm is $O(ed^3)$ where e is number of constraints and d is the size of the greatest domain. Later version of AC algorithm named AC4 [5] algorithm achieves arc consistency in $O(ed^2)$, however, the space complexity increases from $O(e)$ to $O(ed^2)$. The improvement consists in implementing counters during the revision. The counter stores the number of supports that each value in domain has on the constraint.

Another well-known first order consistency is path inverse consistency [6]. Since it is actually $(1, 2)$-consistency, it is stronger than arc consistency.

Definition 3.6 (Path Inverse Consistency) *Let $P = (V, X, C)$ be a CSP and x_a be a valid value of variable $a \in V$. Then x_a is path inverse consistent iff for every two different variables b_1,b_2 from V, $b_1 \neq a$, $b_2 \neq a$, there exists a locally consistent instantiation $\{(a, x), (b_1, y_1), (b_2, y_1)\}, y_1 \in X_{b_1}, y_2 \in X_{b_2}$.*

Example 3.7 *Fig. 5 shows situation where each value is arc-consistent, however, no one is path inverse consistent.*

3.2 Higher-Order Consistency

General arc consistency does not guarantee a solution, hence we need to prove consistency of an instantiation of size greater than one. Higher-order consistencies are stronger first-order consistencies. Here, new constraints are created as an extension of already existing constraints. However, the time complexity of creation of new constraints may increase exponentially. To start with, we introduce a general consistency called $k - consistency$ [7].

Definition 3.8 (k-Consistency) *Let $P = (V, X, C)$ be a CSP problem and $1 < k \leq |V|$ be an integer.*

- *For a set of $k - 1$ variables $A \subseteq V$, a locally consistent instantiation I of A is said be k-consistent iff $\forall v \in V \setminus A$, I can be extended into a locally consistent instantiation of $A \cup v$.*
- *P is said to be k-consistent iff every set A of $k-1$ variables is k-consistent.*

A problem that is k-consistent does not have to be necessarily i-consistent for $1 \leq i < k$ as shown in example 3.9. That is why strong k-consistency is introduced in 3.10.

Example 3.9 *Consider a set of variables* $V = (1, 2, 3)$ *with domains* $X_1 = X_2 = X_3 = \{a, b\}$*. Three binary constraints are defined by relations* $R_{(1,2)} = \{(a, a), (b, b)\}$*,* $R_{(1,3)} = \{(a, a), (b, b)\}$*,* $R_{(2,3)} = \{(a, b), (b, a)\}$*. We can see, that this problem is 2-consistent, however, it is not 3-consistent. It is 2-consistent because every value* $x_a \in X_1 \cup X_2 \cup X_3$ *can be extended over a variable b,b* $\neq a$*, such that* $x_{(a,b)} in X_{(a,b)}$*. E.g.* a_1 *can be extended over* 2 *into* $(a, a)_{(1,2)}$*. However, it is not 3-consistent. E.g. for* $(a, a)_{(1,2)}$ *neither* $\{(1, a), (2, a), (3, a)\}$ *nor* $\{(1, a), (2, a), (3, b)\}$ *is a local consistent instantiation. The example is demonstrated in Fig. 5. It is sometimes called a helix.*

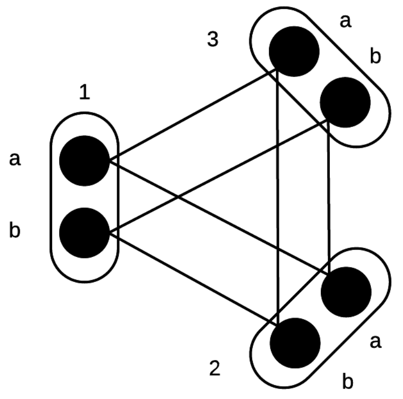


Figure 5 A problem which is 2-consistent, however, it is not 3-consistent.

Definition 3.10 (Strong k-Consistency) *A CSP problem P is strong k-consistent iff it it i-consistent for every* $1 \leq i < k$*.*

Note that a n-consistent CSP problem where n is a number of variables is globally consistent. The n-consistency is desirable since it not only guarantees a solution but also a solution can be found using depth-first search.

While k-consistency extends the set of variables by adding only one variable, a more general concept [8] includes an extension of i variables into a set of $(i + j)$ variables. Analogously to k-consistency, (i, j)-consistency is defined as follows.

Definition 3.11 ((Strong) (i,j)-Consistency) *Let P = (V,X,C) be a CSP and* $0 \leq i < |V|$*,* $i \leq j \leq |V|$ *be two integers such that* $i + j \leq |V|$*.*

- *For a set of i variables* $A \subseteq V$*, a locally consistent instantiation* I *of* A *is said to be k-consistent iff for every subset of j variables* $B \in V \setminus A$*,* I *can be extended into a locally consistent instantiation of* $A \cup B$*.*

- *P is said to be (i,j)-consistent iff every set A of j variables is (i,j)-consistent.*
- *P is said to be strong (i,j)-consistent iff it is (k,j)-consistent for every $0 \le k \le i$.*

It is obvious that k-consistency is $(k-1,1)$-consistency. For a node consistent problem (i.e. domains coincide with unary constraints), arc consistency is equivalent to 2-consistency and $(1,1)$-consistency.

Consistencies mentioned above take a set of variables that relate to each other, i.e. they are involved in the same scope, and prove the consistency. However, we can take two arbitrary variables and compare it with another variable to prove it consistent or not. Such a consistency is called path consistency [9].

Definition 3.12 (Path Consistency) *Let $P = (V, X, C)$ be a CSP.*

1. *A locally consistent instantiation $I = \{(a,i),(b,j)\}$ on P is said to be path consistent iff for any variable $c \in V$ there exists such k that both instantiation $\{(a,i),(c,k)\}$ and $\{(c,k),(b,j)\}$ are locally consistent.*
2. *A pair (a,b) of variables is said to be path consistent iff for all $(i,j) \in X_{(a,b)}$ it is path consistent.*
3. *P is path consistent iff every pair of variables is path consistent.*

Example 3.13 *Consider a set of variables $V = (a,b,c)$ with domains $X_1 = X_2 = X_3 = \{1,2\}$ and a set of constraints $C = \{a \neq b, b \neq c\}$. Variables a and b and pair consistent, however, a and c are not consistent since neither $\{(a,1),(c,2)\}$ nor $\{(a,2),(c,1)\}$ can be extended to a locally consistent instantiation. We can make the problem path consistent by adding a constraint $a = c$.*

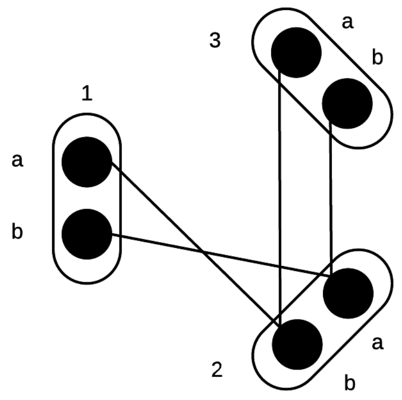


Figure 6 Example 3.13. Variables (a,b) are path consistent, however, (a,c) are not path consistent.

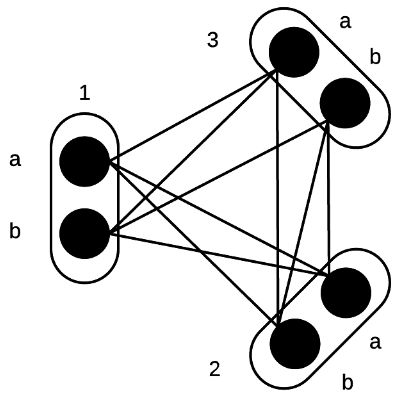


Figure 7 Example 3.14. Instantiation $\{(a,1),(b,1)\}$ is not path consistent.

Example 3.14 *Given a set of variables $V = (a,b,c)$ with domains $X_1 = X_2 = X_3 = \{1,2\}$ and constraints defined by relations $R_{(a,b)} = \{(1,1),(2,1),(2,2)\}$, $R_{(a,c)} = \{(1,1),(2,1),(2,2)\}$, $R_{(b,c)} = \{(1,2),(2,1),(2,2)\}$, we can see that $\{(a,1),(b,1)\}$ can not be extended over c to a locally consistent solution, however, both values 1_a and 1_b lead to a solution. Moreover, removing 1_a or 1_b makes other solution disappear.*

3.2.1 Relation-based consistencies

So far, we have described consistencies that take into account a variable or a set of variables and search for a support on constraints. We can also compare constraints and remove invalid tuples if they are supported on other constraint. Every domain can be considered as a unary constraint. Thereafter, searching for a support for an unary constraint is equivalent to searching for a support for a variable.

Pairwise consistency [10] belongs to the most important relation-based consistencies and it is defined as follows.

Definition 3.15 (Pairwise Consistency) *Let $P = (V, X, C)$ be a CPS.*

- *A constraint $c_A \in C$ on $A \subseteq V$ is said to be pairwise consistent with respect to a constraint $c_B \in C$, $c_A \neq c_B$, on $B \subseteq V$ iff for every locally consistent instantiation of A there exists an extension over a set $B \setminus A$ satisfying the constraint c_B.*
- *P is pairwise consistent iff every constraints $c \in C$ is pairwise consistent with respect to every constraints $\bar{c} \in C$, $\bar{c} \neq c$.*

A normalized binary constraint network that is arc consistent is also pair consistent. Or more precisely, arc consistent constraints sharing only one variable is necessarily pairwise consistent. Hence, pairwise consistency should be applied on constraints with higher arities.

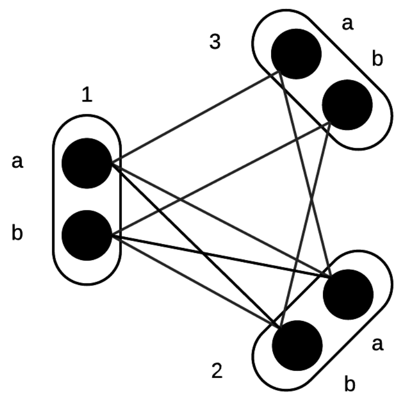


Figure 8 A problem which is arc consistent, however, it is not pairwise consistent.

Example 3.16 *Let X = (1,2,3) be set of variables with domains* $X_1 = X_2 = X_3 = \{a, b\}$ *and* $c_{(1,2)}, c_{(1,2,3)}$ *be two constraints defined by relations* $R_{(1,2)} = (a, b), (b, a)$*,* $R_{(1,2,3)} = (a, a, a), (b, b, b)$ *respectively as shown Fig. 8. This problem is arc consistent, but it is not pairwise consistent. E.g. for* $(a, b)_{(1,2)}$*, both* $(a, b, a)_{(1,2,3)}$ *and* $(a, b, b)_{(1,2,3)}$ *are conflicts.*

It is expensive to revise every pair of constraints, however it can be reduced by listing the set of pairs explicitly. We denote this set by J and say that a problem P is J-consistent iff it is pairwise consistent for every pair in J. In this thesis, we assume $\forall(A, B) \in J, A \subseteq B$.

Definition 3.17 (J-Consistency) *Let* $P = (V, X, C)$ *be a CSP and* $J = (A, B), A \subseteq B$ *be a set of pairs of scopes.* P *is said to be* J*-consistent iff for every* $(A, B) \in P$*,* c_A *pairwise consistent with respect to* c_B*.*

4 WEIGHTED CONSTRAINT SATISFACTION PROBLEM

While a classical CSP helps us decide whether there exists a solution to the problem, we may search for a solution optimizing some objective (cost, time, space, human resources etc.). The aim of weighted constraint satisfaction problem is finding an assignment to a set of variables so that the objective is minimized or maximized. The hard 'YES/NO' constraints are replaced by soft constraint defined by a function mapping each tuple into a value.

Definition 4.1 (Weighted Constraint Satisfaction Problem) *A weighted constraint satisfaction problem (WCSP) is defined by a triplet* (V, X, F) *where* V *is a totally ordered set of* n *variables* $V = (v_1, v_2, \ldots, v_n)$, X *is a set of corresponding domains* $X = (X_{v_1}, X_{v_2}, \ldots, X_{v_n})$ *such that each variable* $v_i \in V$ *takes a value from its domain* X_{v_i}, $F = \{(A, f_A) | A \subseteq V\}$ *is a set of cost functions. Each cost function* (A, f_A) *is defined over a scope* A *by a function* $f_A : x_A \to R$. *The cost function is also called a soft constraint.*

Notation 4.1.1 *Referring to WCSP, we write constraint only instead of soft constraint when no confusion can arise.*

Example 4.2 *Given a gray scale noisy image, we want to extract vertical and horizontal lines from it. We can formulate this task as a weighted CSP. Suppose that white lines cross the black background in the original image. The gray scale image is a two dimensional array of pixels taking values from the grayscale* $0 \ldots 1$ *where* 0 *is black and* 1 *white. For each pixel we have to decide whether it is white (line) or black (background). We represent a pixel as a variable with domain consisting of four values:*

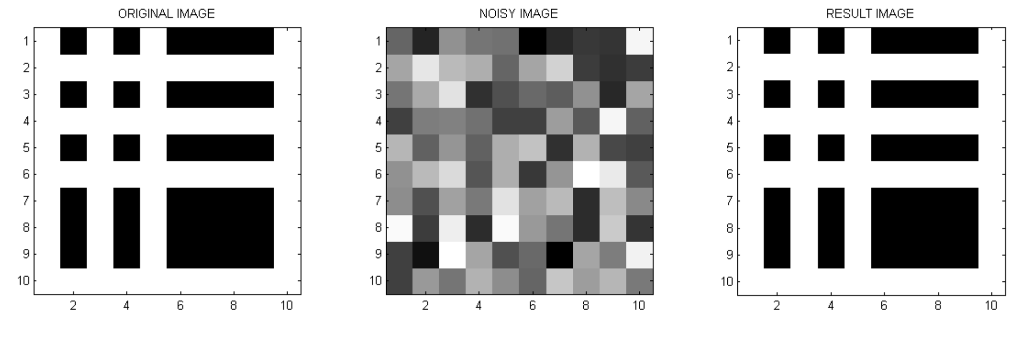
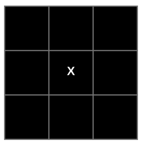
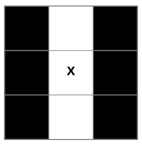


Figure 9 Extracting vertical and horizontal lines from a gray scale noisy image

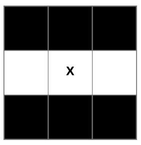
- *a stands for black pixel (Fig. 10a)*
- *b stands for white pixel of horizontal line (Fig. 10b), hence left and right pixels are white and pixels above and below are white.*
- *c stands for white pixel of vertical line (Fig. 10c), hence pixels above and below are white, left and right pixels are black.*
- *d stands for white pixel of line cross (Fig. 10d), hence the 4-neighbourhood [1] is white.*



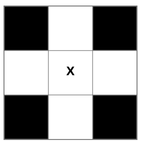
a) Black pixel



b) White pixel of horizontal line



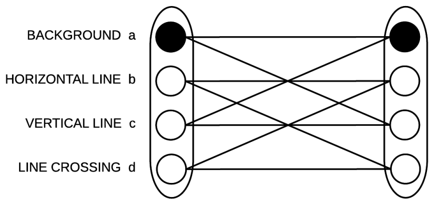
c) White pixel of horizontal line



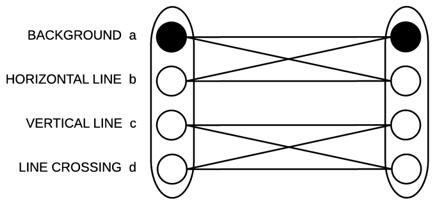
d) White pixel of line cross

Figure 10 All possible position of a pixel

A binary "YES/NO" constraint is defined for every pair of neighbouring pixels and the constraint differs for vertically and horizontally adjacent pixels. All possibles tuples are displayed in Fig. 11.



a) Allowed combination of tuples for pixel horizontally adjacent



b) Allowed combination of tuples for pixel vertically adjacent

Figure 11 All possible combinations of pixels

Finally, we can state the cost functions. Let p_i be a value of pixel i. For each variable i, the unary cost function f_i is given by

$$f_i(x_i) = \begin{cases} p_i & \textit{if } i = a \\ 1 - p_i & \textit{otherwise} \end{cases}$$

For each pair (i, j) of adjacent pixels, the binary function f_A is defined by

$$f_A(x_A) = \begin{cases} 0 & \textit{if the tuple } x_A \textit{ is allowed by } c_A \\ \infty & \textit{otherwise} \end{cases}$$

[1] In image processing, 4-neighbourhood expresses the way how pixel relate to each other. 4-neighbourhood of a pixel p includes pixels that are vertically and horizontally connected to it, hence maximal number is 4.

As already mentioned, a assignment is chosen to minimize or maximize an objective. We define the total cost (also called valuation or energy) as follows.

Definition 4.3 (Total Cost) *Given a WCSP P = (V,X,F), the total cost g of assignment $x_V \in X_V$ is given by*

$$g(x_V) = \sum_{f_A \in F} f_A(\pi_A(x_V))$$

In WCSP, the aim is to find an assignment x_V minimizing the total cost.

$$x_V = \operatorname*{argmin}_{x_V \in X_V} \sum_{f_A \in F} f_A(\pi_A(x_V))$$

4.1 Equivalent-preserving transformations

Along with the soft constraints, we need to extend the operations on them. The basic operations are combination and projection. Similarly to join, combination merges several constraints into a newly created constraint. Projection shifts weight from a constraint to a newly created constraint. The equivalence is kept since the new constraint holds the information.

Definition 4.4 (Combination) *Let $P = (V, X, F)$ be a WCSP. Given two functions f_A and f_B, combination, denoted by $f_A \bowtie f_B$, is a function g given by $g(x_C) = f_A(\pi_A(x_C)) + f_B(\pi_B(x_C))$ where $C = A \bowtie B$.*

c_1

a_1	cost
a	1
b	2

c_2

a_1	a_2	cost
a	a	0
a	b	3
b	a	2
b	b	5

$c_3 = c_1 \bowtie c_2$

a_1	a_2	value
a	a	1
a	b	4
b	a	4
b	b	7

Figure 12 Example of combination of two soft constraints

Definition 4.5 (Projection) *Let $P = (V, X, F)$ be a WCSP. Given a cost function f_B and a set of variables $A \subseteq B$, projection of f_B over A, denoted by $\pi_A(f_B)$, is a function g given by $g(x_A) = \sum_{x_B \in X_B | \pi_A(x_B) = x_A} f_B(x_B)$.*

f_A

a_1	a_2	a_3	cost
a	a	a	2
a	a	b	3
a	b	a	1
a	b	b	0
b	a	a	4
b	a	b	0
b	b	a	0
b	b	b	1

$\pi_{(a_1,a_2)}(f_A)$

a_1	a_2	cost
a	a	5
a	b	1
b	a	4
b	b	1

$\pi_{a_1}(f_A)$

a_1	cost
a	6
b	5

$\pi_{\emptyset}(f_A)$

cost
11

Figure 13 Example of soft constraint projection

Example 4.6 *Consider a network of including two variables* 1 *and* 2 *with domains* $X_1 = X_2 = \{(a, b)\}$ *and a binary cost function* $f_{(1,2)}$ *assigning* α *to pairs* (a, a) *and* (b, b) *and* 2α *to* (a, b)*. The figure 14 shows a possible sequence of equivalence preserving transformations. First,* $f_{(1,2)} \bowtie f_2$ *is projected on* a_1*. Simultaneously, we modify* $f_{(1,2)}$ *by subtracting* α *from pairs* (a, a) *and* (a, b)*. The same process is applied on* b_2*. The equivalence is kept since the weight has been neither added nor lost and the information has been transferred from* $f_{(1,2)}$ *to* f_1 *and* f_2*. On the other hand, Fig. 15 demonstrates an example of operations which break the equivalence.*

As example 4.6 shows, it turns out to be useful to define operation extracting cost functions. We call it extraction and it is defined as follows.

Definition 4.7 *Let* $P = (V, X, F)$ *be a WCSP. Given two functions* f_A *and* f_B*, extraction, denoted by* $f_B \ominus f_A$*, is a function* g *given by* $g(x_C) = f_B(\pi_B(x_C)) - f_A(\pi_A(x_C))$ *where* $C = A \bowtie B$*.*

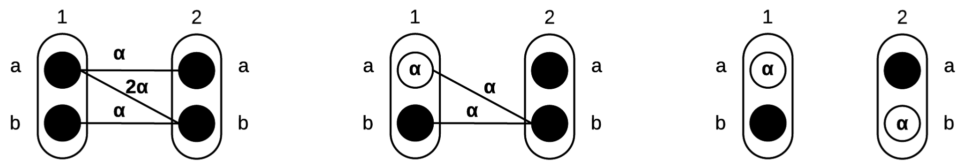


Figure 14 Sequence of equivalence preserving transformations

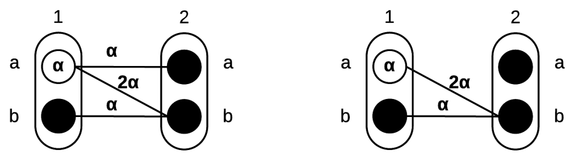


Figure 15 Operations where equivalence is broken

Soft constraint operations are used to enforce local consistencies. They are combined to enable the flow of weights in the network. The process of shifting costs between constraints consists of two steps. First, a new cost function is created. Then it is extracted from the source functions and added to the target function. To preserve the equivalence, (K, Y)-equivalence-preserving rule is defined.

Definition 4.8 ((K, Y)-Equivalence-Preserving Rule) *Let $P = (V, X, F)$ be a WCSP, $K \subset F$ be a set of functions and $Y \subset V$ be a set of variables. The (K, Y)-equivalence-preserving rule (EP rule) consists in*

1. *removing K from the network,*
2. *adding $\pi_Y(\bowtie K)$ and $(\bowtie K) \ominus \pi_Y(\bowtie K)$ to the network.*

The rule explains how to move cost from a set of constraint K to scope Y. Informally, we subtract the cost from functions in K and add it to the function defined over A. In the following section, we will show how equivalence-preserving transformations help eliminating local inconsistency.

4.2 Soft Consistency

In this section we briefly introduce soft consistencies and how to achieve them using equivalent preserving transformations.

Definition 4.9 (Soft Node Consistency) *Let $P = (V, X, F)$ be a WCSP and ϵ be a constant. We call a variable $a \in V$ soft node consistent iff for every variable $a \in V$,*

- $f_a(x_a) \leq \epsilon$.

The node consistency can be obtained by projecting the the minimum from each constraint to $\emptyset$. For constraint of any arity, we introduce soft general arc consistency and soft J-consistency.

Definition 4.10 (Soft General Arc Consistency) *Let $P = (V, X, F)$ be a node consistent WCSP and ϵ be a constant. We call a variable $a \in V$ generally arc consistent with respect to constraint f_A iff for every value $x_a \in X_a$,*

- $f_a(x_a) \bowtie f_A(x_A) \bowtie (\bowtie_{x_B \in X_B | \pi_B(x_A) = x_B} (f_B(x_B))) \leq \epsilon$.

Definition 4.11 (Soft J-Consistency) *Let $P = (V, X, F)$ be a WCSP and $J = \{(A, B) | A \in V, B \in V, A \subseteq B\}$ be a set of pairs of scopes. A constraint $f_A \in F$ on $A \subseteq V$ is said to be soft pairwise consistent with respect to constraint $f_B \in F, f_A \neq f_B$, on $B \in V$ iff for every valid $x_A \in X_A$ such that $f_A(x_A) \leq \epsilon$*

- $\exists x_B \in X_B$ *such that* $\bowtie f_B(x_B) \bowtie \pi_C(\bowtie_{x_C \in X_C | \pi_C(x_B) = x_C} (f_C(x_C))) \leq \epsilon$

P is said to be J-consistent iff if every pair of scopes in J is pairwise consistent.

We can achieve soft J-consistency by repeating (B, A) rules where $(A, B) \in J$. However, by chaotic application of the rule does not have to lead to J-consistency and the sequence of the soft operations should by planned. This is reached by techniques used in classical CSP to which a WCSP is transformed. A CSP derived from WCSP is called Bool(P).

Definition 4.12 (Bool(P)) *Let P = (V,X,F) be a WCSP. Bool(P) is classical CSP (V,X,C) where* $c = (A, R_A) \in C$ *iff* $\exists(A, f_A) \in F$ *and* R_A *is given by* $x_A \in X_A(x_A \in R_A \Leftrightarrow f_A(x_A) = 0)$.

Enforcing J-consistency on Bool(P) leads to deletion of allowed tuples in constraints. When all tuples in a constraint in Bool(P) are disallow, there exists a sequence of soft consistency operations that leads to increase in the lower bound. Nullary cost function $f_\emptyset$, defined over an empty set, constitutes the lower bound on any solution.

Definition 4.13 (J-Consistent Closure) *Given a network P = (V,X,C), a J-consistent closure of P is a network obtained by deleting all J-inconsistent tuples from C.*

Definition 4.14 (Virtual J-Consistency) *A problem P is said to be virtual J-consistent if there exists a non-empty J-consistent closure of Bool(P) that is every constraint in Bool(P) contains at least one valid tuple.*

In the following section, we introduce an algorithm that applies virtual J-consistency on WCSP.

5 VIRTUAL J-CONSISTENCY ALGORITHM

The algorithm in [3] deals with arc consistency which is applied on Bool(P). We modified this algorithm to detect J-consistencies. J-consistency proves consistent only constraints listed in J and is more general than arc consistency. It can be weaker or stronger than AC depending on number of pairs in J. The algorithm enforces J-consistency to shift the λ amount from its source to the nullary cost function representing the lower bound.

The VJC algorithm consists of three phases as follows.

1. Search for the J-consistent closure of Bool(P). If the closure exist, stop the algorithm. Otherwise keep track of all deleted inconsistent values and the cause of their deletion.
2. Compute λ amount to be added to the lower bound. Starting from the constraint i_0 whose all tuple were disallowed, track back to the source of deletion. Select the lowest value of all sources.
3. Apply the sequence of equivalence-preserving transformations. Shift the amount λ from its source toward the nullary cost function $f_\emptyset$.

5.1 First phase: J-consistency algorithm

During the first phase of the algorithm, we detect J-inconsistency in Bool(P). Since we delete tuples in relations rather than values from domains of variables, we define a Boolean function $\bar{f_A} : X_A \to \{0, 1\}$ by

$$\bar{f_A}(x_A) = \begin{cases} 0 & \text{if } \bigvee_{(C,B)\in J} \bar{f_C}(x_C) \vee killer_{x_A} > 0 \vee f_A(x_A) > \epsilon \\ 1 & \text{otherwise} \end{cases}$$

The process of detecting inconsistencies is based on the AC3 algorithm. First, we initialize a stack P containing all couples of constraints we want to prove J-consistent, hence the initial stack P is set J. Then we iteratively revise all couples $j \in P$ until P is empty. The revise procedure consists in iterating all tuples $x_A \in X_A$ (X_A is the Cartesian product of current domains) and searching for their support. If a support does not exist, x_A is deleted. To record the cause of the deletion, $killer_{x_A}$ is set to j. $killer$ is used only for x_A where $(A, B) \in J$.If any of following conditions is satisfied, $x_B \in X_B$ is determined to be conflict:

a) $f_B(x_B) > \epsilon$. The tuple x_B is not consistent on f_B.
b) $\bigvee_{(C,B)\in J} \bar{f_C}(x_C) \wedge f_B(x_B) \leq \epsilon$. The tuple x_B is consistent on f_B, however, there exists a scope C included in B that invalidated $x_B \in X_B$

Algorithm 1: Phase 1: Instrumented-JC

```
Function instrumented-JC()
    $P \leftarrow J$
    while $P \neq \emptyset$ do
        $(A, B) \leftarrow P.pop()$
        if revise($A$,$B$) then
            if $X_A = \emptyset$ then return $A$
            else
                $P \leftarrow P \cup \{(\bar{A}, B) | (A, B) \in J, (\bar{A}, B) \in J,\ A \neq \bar{A}\} \cup$
                $\{(\bar{A}, A) | (\bar{A}, A) \in J\}$
    return $\emptyset$
Function revise($A$,$B$)
    $changed \leftarrow false$
    for $x_A \in X_A$ s.t. $\bar{f}_A(x_A)$ do
        if $\nexists x_B \in X_B$ s.t. $\pi_A(x_B) = x_A \wedge \bar{f}_B(x_B) = 1$ then
            $killer_{x_A} \leftarrow (A, B)$
            $nbOfDeleted{+}{+}$
            $delete_{x_A} = nbOfDeleted$
            $changed \leftarrow true$
    return $changed$;
```

c) $\exists C$ such that $(B, C) \in J$ and $killer_{x_B} > 0$. The tuple x_B is consistent on f_B, however, there exists a scope $C, C \subseteq B$ such that x_B is killed because of lack support on f_C.

In addition to setting the *killer*, we also record the order of deletion by setting *delete*. In next phase, this helps us to identify the killer and build the minimal structure explaining the wipe-out.

If revise deletes any tuple, some constraints have to be proved consistent again. Suppose that X_A has been modified because of lack support on X_B. This means that any constraint whose scope is involved in B may have lost its support on X_B and it need to be added to the stack P unless already inserted. Moreover, any tuple $x_{\bar{A}}, (\bar{A}, A) \in J$ has to be revised again.

Example 5.1 *We have a set of variables* $V = (1, 2, 3, 4)$ *with domains* $X_1 = X_2 = X_4 = a, b$ *and* $X_3 = \{a, b, c\}$. *The cost functions are given as follows.*

- *unary cost functions:*
  - $f_1(a) = 1, 0$ *otherwise.*
  - $f_2 = 0$.
  - $f_3(c) = 1, 0$ *otherwise.*
  - $f_4 = 0$.
- *binary cost functions*
  - $f_{(1,3)}(b, b) = f_{(1,3)}(b, c) = 0, 1$ *otherwise.*
  - $f_{(1,4)}(a, b) = f_{(1,4)}(b, a) = 0, 1$ *otherwise.*
  - $f_{(3,4)}(a, a) = f_{(3,4)}(b, a) = f_{(3,4)}(c, b) = 0, 1$ *otherwise.*

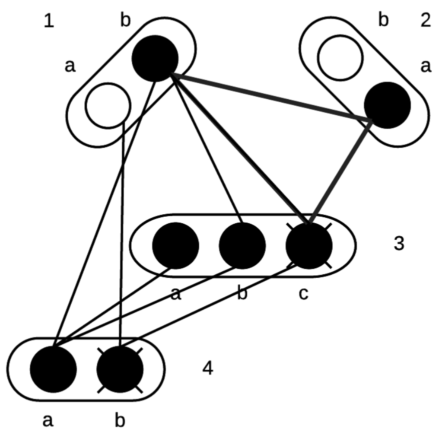


Figure 16 Detecting conflicts

Figure 17 Graphical representation of example 5.1

- *ternary cost functions:*
  - $f_{(1,2,3)}(b, a, c) = 0, 1$ *otherwise.*

The example is demonstrated in Fig. 16. Suppose that value c_3 has been killed because of some other inconsistency. Now we want to revise (A, B) where $A = (1)$ and $B = (1, 3)$. Value a_1 does not belong to the domain X_a since $f_1(a) = 1$. b_1 is killed because all x_B where $\pi_1 x_B = b$ are conflicts.

- *$x_{(1,3)} = (b, a)$ is conflict because $f_{(1,3)}(b, a) = 1$.*
- *$x_{(1,3)} = (b, b)$ is conflict because it was killed, since it has no support on cost function $f_{(1,2,3)}$.*
- *$x_{(1,3)} = (b, c)$ is conflict because f_3 is an unary cost function involved in binray function $f_{(1,3)}$ and $(3, a)$ has been killed.*

5.2 Second phase: Computing λ

The JC algorithm runs until all constraints are revised. If any domain has been wiped out, Bool(P) is not J-consistent. Suppose that X_{A_0} has been wiped out then we compute the minimal λ amount that can be subtracted from some cost function in the network to remove the cause of the wipe-out. Starting from A_0, we build a structure explaining the wipe-out. This structure forms a directed acyclic graph (DAC) whose nodes represent tuples in cost functions. The edges explain the direction in which λ amount is computed. An edge leading from x_A to x_B is labelled by $j \in J$ that caused deletion of x_A. Note that label can be either $j = (A, B)$ or $j = \{(A, C), (C, B)\} \in J$ There are three types of nodes in DAG:

- The root node of the DAG symbolizes the nullary function.

Algorithm 2: Phase 2: Computing λ

```
Procedure Compute$\lambda$()
       Initialize all $k$,$k_J$ to 0, $\lambda \leftarrow \infty$
 1     $A_0 \leftarrow$ instrumented-JC()
 2     if $A_0 = \emptyset$ then return
 3     for $x_{A_0} \in X_{A_0}$ do
 4         if $f_{A_0}(x_{A_0}) \leq \epsilon$ then
 5             $k(x_{A_0}) \leftarrow 1$, $M(x_{A_0}) \leftarrow true$
 6             $Q$.push($delete_{x_{A_0}}$, $(x_{A_0}, A_0)$)
 7         else $M(x_{A_0}) \leftarrow false, \lambda \leftarrow min(\lambda, f_{A_0}(x_{A_0}))$

 8     while $Q \neq \emptyset$ do
 9         $(x_A, A) \leftarrow Q$.pop()
10         $j \leftarrow killer_{x_A}$                                ▷ let $j = (A, B)$
11         $R$.push$(x_A, A)$
12         for $x_B \in X_B$ $s.t.$ $\pi_A(x_B) = x_A$ do
13             if $f_B(x_B) > \epsilon$ then
14                 $k(x_B) \leftarrow k(x_B) + k(x_A)$
15                 $\lambda \leftarrow$ min $(\lambda, \frac{f_B(x_B)}{k(x_B)})$
16             else
17                 if $killer_{x_B} > 0 \ \wedge \ delete_{x_B} < delete_{x_A}$ then
18                     if $\neg M_{x_B}$ then
                           $k(x_B) \leftarrow k(x_B) + k(x_A)$
19                         $Q$.push($delete_{x_B}$, $(x_B, B)$), $M_{x_B} \leftarrow true$
20                 else
21                     for $x_{\bar{A}} \in X_{\bar{A}}$ $s.t.$ $(\bar{A}, B) \in J$, $x_{\bar{A}} = \pi_{\bar{A}}(x_{\bar{A}})$,
                       $delete_{x_{\bar{A}}} < delete_{x_A}$ do
22                         if $k(x_A) > k_J(x_{\bar{A}})$ then
23                             $k(x_{\bar{A}}) \leftarrow k(x_{\bar{A}}) + k(x_A) - k_J(x_{\bar{A}})$
24                             $k_j(x_{\bar{A}}) \leftarrow k(x_A)$
25                             if $f_{\bar{A}}(x_{\bar{A}}) \leq \epsilon$ then
26                                 if $\neg M_{x_B}$ then
27                                     $Q$.push($delete_{x_B}$, $(x_B, B)$), $M_{x_B} \leftarrow true$
28                             else
                                   $\lambda \leftarrow min(\lambda, \frac{f_{\bar{A}}(x_{\bar{A}})}{k(x_{\bar{A}})})$
```

- Inner nodes stand for the tuples whose weight is less or equal ϵ, however, they have been removed because of inconsistency.
- Leaf nodes represent tuples with weight greater than to ϵ. λ is computed as the minimum over the leaf node weights.

Example 5.2 *In example 5.1, tuple $(b, c)_{(1,2)}$ was deleted because of lack support on $f_{(1,2,3)}$). Tuple b_4 was killed because of lack support on $f_{(1,4)}$. Consequently, c_3 was killed by $j = ((3), (3, 4)$ and finally, b_1 killed by $j = ((1), (1, 3))$,*

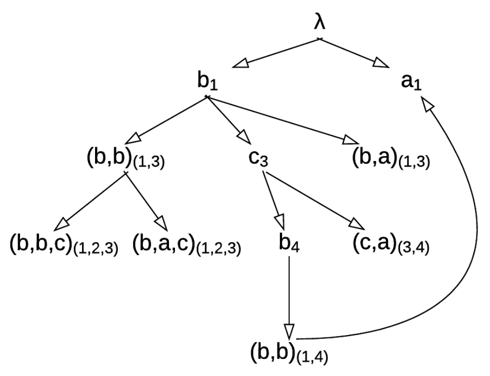


Figure 18 Computing λ in example 5.1

hence $A_0 = (1)$. *The DAG is demonstrated in Fig 18.*

To build the DAG, we use a priority queue Q in which tuples are sorted in descending order of their deletion. The order has to be kept since there may exist more paths leading from the root to an inner node. First, tuples $x_{A_0}, f_{A_0}(x_{A_0}) \leq \epsilon$ are inserted into Q. Then we iteratively process each tuple in Q (line 8). For each x_A, we find out and discuss all causes of its deletion (line 12). Analogously to searching for the support for x_A, there can arise three causes of deletion. Given a killed tuple x_A, its killer $j = (A, B)$ and a tuple $x_B, \pi_A(x_B) = x_A$, the causes can be as follows.

a) (line 13) $f_B(x_B) > \epsilon$. This tuple is not ϵ consistent, hence it constitutes a leaf node and λ is computed from it.
b) (line 17) x_B is killed and therefore it is no longer a support for x_A. Moreover, we have to check the order of deletion to ensure that x_B invalidated x_A. x_B is inserted into Q to find next λ sources.
c) (line 20) x_B is not a support because there exists a tuple $x_C, (C, B) \in J$ that caused deletion of x_B. We iterate all the tuples x_C that invalidated x_B. Tuples $x_C, f_C(x_c) > \epsilon$ compose leaves of the DAG, hence they influence λ. Consistent tuples are added to Q and the explanation of their deletion will help to explain the wipe-out later.

It is obvious that one tuple can cause deletion of several tuples, therefore λ is sent to A_0 through multiple paths. Here, counters on tuples are introduced. Each tuple explaining the wipe-out records the number of requests for the λ amount and the direction in which the amount will be send. The counter $k_j(x_B)$ indicates how many tuples were deleted because of j. The counter $k(x_B)$ stores the total number of deleted tuples. If there are more tuples $x_A \in X_A$ killed by j that demand some amount from x_B, x_B remembers only the one with highest number of requests since one call of extension sends the amount to all $x_A \in X_A$. Hence, if there is a new request from a tuple deleted by j that is higher than the yet highest one (line 22), both k and k_j have to increase accordingly. Note that $k(x_B) = \sum_{j \in J, j=(A,B)}(k_j(x_A))$

Example 5.3 *Consider the example 5.1. The inconsistency of $f_1(a_1)$ killed both tuples a_1 directly and b_1 indirectly. λ is equally split and a half of λ amount is send via $f_{(1,4)}$,f_4,$f_{(3,4)}$,f_3,$f_{(1,3)}$,f_1 to $f_\emptyset$ and the other half is projected to $f_\emptyset$ directly.*

5.3 Third phase: Applying equivalence-preserving transformations

Suppose that $A \subset B$ holds for every $(A, B) \in J$. Equivalence-preserving transformation send weights from cost functions with higher arity to cost functions to lower arity and vice versa. Weights can be sent between f_A and f_B if and only if $(A, B) \in J$. The project operation sends weights from a cost function f_B to a cost function of lower arity f_A. The extend operation extends weights from f_A to f_B. UnaryProject subtracts energy from a cost function and adds it to the nullary cost function $f_\emptyset$ representing the lower bound of any solution.

Example 5.4 *Given a cost function f_A, $\forall x_A \in X_A, f_A(x_A) \geq \alpha$, we can call UnaryProject until $\exists x_A, f_A(x_A) = 0$. Calling UnaryProject function on cost functions and deducing the minimum value from them leads to the inevitable increase of the lower bound on the solution.*

In the third phase, the direction of edges in DAG is swapped and λ is sent from the leaf nodes toward the root of DAG. Iterating the inner nodes, we send adequate cost from the leaf to the inner node (line 5) if it has non zero k_j. Then we can project λ amount.

Example 5.5 *After applying equivalence preserving transformations in example 5.1, the cost functions are as follows.*

- *unary cost functions:*
  - $f_1(a) = 0$ *otherwise.*
  - $f_2 = 0$.
  - $f_3(c) = 1, 0$ *otherwise.*
  - $f_4 = 0$.
- *binary cost functions*
  - $f_{(1,3)}(b, b) = f_{(1,3)}(b, c) = 0, f_{(1,3)}(a, c) = 1.5$ *and 1 otherwise.*
  - $f_{(1,4)}(a, b) = f_{(1,4)}(b, a) = 0, f_{(1,4)}(a, a) = 1.5$ *and 1 otherwise.*
  - $f_{(3,4)}(a, a) = f_{(3,4)}(b, a) = f_{(3,4)}(c, b) = 0, f_{(3,4)}(a, a) = f_{(3,4)}(b, a) = 1.5$ *and 1 otherwise.*
- *ternary cost functions:*
  - $f_{(1,2,3)}(b, a, c) = 0, f_{(1,2,3)}(b, a, b) = f_{(1,2,3)}(b, b, b) = 0.5$ *and 1 otherwise.*

Algorithm 3: Phase 3: Applying equivalence-preserving transformations

```
Procedure ApplyEPT()
1     while $R \neq \emptyset$ do
2         $(x_A, A) \leftarrow R$.pop()
3         $j \leftarrow killer_{x_A}$                                   ▷ let $j = (A, B)$
4         for $\bar{j} \in J, x_{\bar{A}} \in X_{\bar{A}}$ s.t. $\bar{j} = (\bar{A}, B),\ k_{\bar{j}}(x_{\bar{A}}) > 0$ do
5             extend($x_{\bar{A}}$,$\bar{j}$,$\lambda \times k_{\bar{j}}(x_{\bar{A}})$)
6             $k_{\bar{j}}(x_{\bar{A}}) \leftarrow 0$
7         project($j$,$x_A$,$\lambda \times k(x_A)$)
8     unaryProject($i_0$,$\lambda$)

9 Function project($j$,$x_A$,$\alpha$)
10    $f_A(x_A) \leftarrow f_A(x_A) + \alpha$
11    for $x_B \in X_B$ s.t. $(A, B) = j, \pi_A(x_B) = x_A$ do
12        $f_B(x_B) \leftarrow f_B(x_B) - \alpha$

13 Function extend($x_A$,$j$,$\alpha$)
14    project($j$,$x_A$,$-\alpha$)

15 Function unaryProject($A$,$\lambda$)
16    for $x_A \in X_A$ do
17        $f_A(x_A) = f_A(x_A) - \alpha$
18    $f_\emptyset \leftarrow f_\emptyset + \alpha$
```

6 REUSING INFORMATION FROM PREVIOUS ITRATIONS

We have observed that many constraints that have already been revised remain stack P, therefore revised again in the next iteration. Some constraints are not influenced by applying EP transformations, hence they do not need to be revised unless they are reinserted into the stack P during the arc consistency algorithm. Hence, constraints influenced by EP transformations can be revived and the information about the killed tuples can be reused in the next iteration.

The algorithm [3] describes one iteration. However, several iteration are required in most cases. We propose a modification of the algorithm for n-ary soft constraints that uses effectively the information from the previous iterations. Since the algorithm propagates general arc consistency and GAC is a special case of J-consistency, where $\forall(A, B) \in J$, $|A| = 1$, we will work with the codes introduced in section pseudocode. Moreover, the algorithm in section pseudocode builds the DAG dynamically and requires the killer information only. In this section, we will suppose that the unary scopes coincide with domains, hence by deleting a value from a domain we mean deleting a tuple from an unary constraints.

Example 6.1 *We try to eliminate the redundant calls of* `revise`*. Consider a network with a large amount of variables and functions, constraints respectively, defined over them. During the* `instrumented-JC` *we have already revised a great deal of constraints in stack P, when a domain of a variable i_0 is wiped out because of lack support on c_B. We compute λ and apply EP transformations. Because the deletion of i_0 was only influenced by c_B, we project the cost from f_B to f_{i_0} and then to $f_\emptyset$. By initializing the structure, we discard the results from the previous iteration, hence we have to revise all constraints again although they were not were not influenced by EP transformations.*

Motivated by example 6.1, we introduce the process of reviving the values from domains. In 6.1, if we revive i_0 and revise constraints including the variable i_0, we can process the next iteration with the modified stack P instead of initializing a new one.

While applying EP-transformations, we have to determine which values we can revive. Every value in DAG was deleted because of some inconsistency. When the source of the inconsistency is removed by shifting weights to the nullary cost function, all values influenced by it have to be revived. The revival is based on following observations.

- Let x_A be a leaf node in DAG, hence $f_A(x_A) > \epsilon$. If the cost of x_A decreases after an iteration to be less or equal ϵ, the node comes alive. Hence the killer and the order of deletion is set zero.
- If a x_A comes alive, tuples killed by inconsistency in f_A are revived as well. Tuples that are not involved in DAG can be revived immediately after the revival of x_A.
- A tuple x_A can be killed by (A, B) because of two reasons:
  - $f_B(x_B) > \epsilon, \pi_A(x_B) = x_A$
  - $x_B, \pi_A(x_B) = x_A$ is inconsistent becausere associated

Example 6.2 *Let* $V = (1, 2, 3, 4)$ *be a set of variables with domains* $X_1 = X_2 = X_3 = X_4 = \{a, b, c\}$. *Let be cost functions defined as follows.*

- *unary functions* $\{f_1, f_2, f_3, f_4\}$*:* $f_1(b) = f_1(c) = 10$*,* $f_3(a) = 1$ *and 0 otherwise.*
- *binary functions* $\{f_{(1,2)}, f_{(2,3)}, f(3, 4)\}$*:* $f_{(1,2)}(a, a) = f_{(1,2)}(a, b) = f_{(2,3)}(a, a) = f_{(2,3)}(b, b) = f_{(2,3)}(c, c) = f_{(3,4)}(a, a) = f_{(3,4)}(b, b) = f_{(3,4)}(c, c) = 0, f_{(2,3)}(a, b) = 1$ *and 10 otherwise.*

During the AC algorithm domain of variable 1 *was wiped-out and killer structure is stored as follows.*

- $killer_{a_4} = (3, 4)$
- $killer_{b_3} = (2, 3)$
- $killer_{b_4} = (3, 4)$
- $killer_{a_2} = (2, 3)$
- $killer_{b_2} = (2, 3)$
- $killer_{a_1} = (1, 2)$

The example is demonstrated in Figure 19. The edges represent binary cost functions and circles stand for values. During the second phase, we determine $\lambda = 1$. *The inner nodes of DAG (i.e. M is set true) are* a_1, a_2 *and* b_2. *Then, the EP transformations are executed as described bellow.*

- `extend(`a_3*,(2,3),1*`)`. *The cost of* a_3 *decreases, thus* $f_3(a) = 0$ *and so* a_3 *can be revived. Since* a_3 *caused deletion of* a_4*, it can be revived as well.* a_3 *also killed* a_2 *and* b_2*, however, these tuples belong to DAG (i.e.* $M(a_2) = M(a_3) = true$*) and we will need the killer information later.*

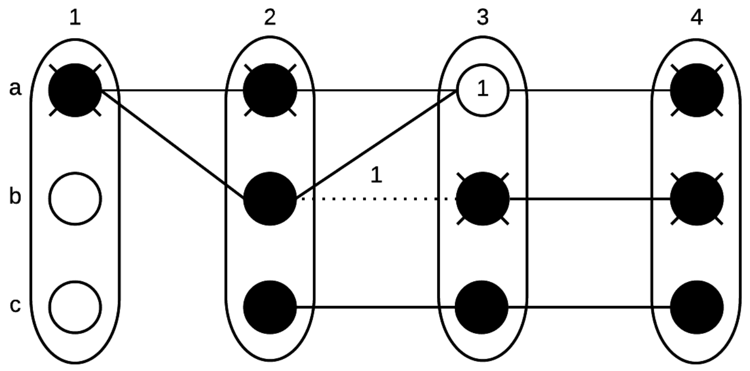


Figure 19 Reviving killed values

- `project(`*(2,3)*,a_2,*1*`)`.
- `project(`*(2,3)*,b_2,*1*`)`.
- `extend(`a_2,*(1,2)*,*1*`)`. *Since* $f_a(a_2) = 0$ *and* a_2 *has been already processed, it can be revived.*
- `extend(`b_2,*(1,2)*,*1*`)`. *Since* $f_a(b_2) = 0$ *and* b_2 *has been already processed, it can be revived. Moreover,* b_3 *that is not a part of DAG can be revived as well.*
- `project(`*(1,2)*,a_1,*1*`)`
- `unaryProject(`*1,1*`)`. *Finally,* a_1 *can be revived.*

We introduce the modification of the third phase of VJC algorithm. While EP transformations are applied, killed values are processed in the reverse order due to stack R. When any λ amount is extended from x_C, we check if it can be revived (line 7. While reviving, killer and order of deletion is set zero. However, value x_C does not necessarily satisfy all constraints since it can be

Algorithm 4: Phase 3: Applying equivalence-preserving transformations and reviving tuples

```
Procedure ApplyEPT()
1     while $R \neq \emptyset$ do
2         $(x_A, A) \leftarrow R$.pop()
3         $j \leftarrow killer_{x_A}$                                  ▷ let $j = (A, B)$
4         $M_{x_A} \leftarrow false$
5         for $\bar{j} \in J, x_{\bar{A}} \in X_{\bar{A}}$ s.t. $\bar{j} = (\bar{A}, B), \bar{A} \neq A$, $k_{\bar{j}}(x_{\bar{A}}) > 0, \neg M(x_{\bar{A}})$ do
6             extend($x_{\bar{A}}$,$\bar{j}$,$\lambda \times k_{\bar{j}}(x_{\bar{A}})$)
7             if $f_{\bar{A}}(x_{\bar{A}}) \leq \epsilon \wedge \bar{A} \neq A_0$ then
8                 revive($x_{\bar{A}}$)
9             $k_{\bar{j}}(x_{\bar{A}}) \leftarrow 0$
10            $k(x_{\bar{A}}) \leftarrow 0$
11        project($j$,$x_A$,$\lambda \times k(x_A)$)
12        for $x_B \in X_B$ s.t. $\pi_A(x_B) = x_A$ do
13            $k_B(x_B) \leftarrow 0$
14    unaryProject($i_0$,$\lambda$)
15    for $x_{i_0} \in X_{i_0}$ s.t. $f_{i_0}(x_{i_0}) \leq \epsilon$ do revive($x_{i_0}$)
16    for $x_{i_0} \in X_{i_0}$ do $k(x_{i_0}) = 0$

Function revive($x_A$)
17    $killer_{x_A} \leftarrow 0$
18    $delete_{x_A} \leftarrow 0$
19    for $B$ s.t. $(A, B) \in J$ do
20        $P \leftarrow P \cup (A, B)$
21    for $(A, B) \in J, (\bar{A}, B) \in J, x_{\bar{A}} \in X_{\bar{A}}$ s.t. $A \neq \bar{A}$ do
22        if $killer_{x_{\bar{A}}} = (\bar{A}, B) \wedge \neg M(x_{\bar{A}})$ then
23            revive($x_{\bar{A}}$)
```

killed because of some other inconsistency, hence we insert all constraints in which x_C is involved into the stack P. Then all variables sharing any constraint c with x_C are checked. If any value x_D of a variable outside of DAG was killed by x_C, i.e. $M(x_D)$ is false, it is revived as well. Then we recursively revive all killed values. When all the unary costs required by j are extended, the amount is projected to x_A and the counter is reset. Finally, the λ amount is sent to $f_\emptyset$ and values with $f_{A_0} \leq \epsilon$ are revived.

In VAC, a stack Q with all killed values is created during the first phase. This structure does not enable reviving the values, hence it is replaced by dynamic creation of DAG using the priority queue. Although insertion a deletion from priority queue is $O(\log(n))$, it contains only few elements.

7 RESULTS

The pseudocode was implemented in MATLAB to support the development of the algorithm. Generally, MATLAB implementation shows lower performance results than implementation in higher languages such as C++ or Java.

The tests were running on a 2.4 GHz Intel(R) Core(TM) i5-2430M with 8 GB using MATLAB R2013b.

7.1 J-Consistency Evaluation

First test indicates how the size J influences time and the result. A set of binary problems with 400 variables and 1160 constraints (400 unary and 760 binary) was randomly generated. On every problem arc consistency was applied that is 1560 pairs of unary and binary constraint are prove consistent. Then the size of J was decreased to 1500, 1300, 1200, 900, 600 and 300 randomly selected pairs. The results of weaker J-consistency are compared to arc consistency. Fig. 20 displays lower bound, number of iterations and time relatively to the arc consistent solution. The big difference in processing time between $|J| = 1500$ and $|J| = 1560$ is caused by ordering of the set J. The algorithm turns out to run faster when the set J is randomly ordered. This observations leads to the heuristic that may speed up the algorithm: to shuffle the stack P. It can be shuffled at beginning of every iteration not to repeat the same sequence of revise functions as in the last iteration.

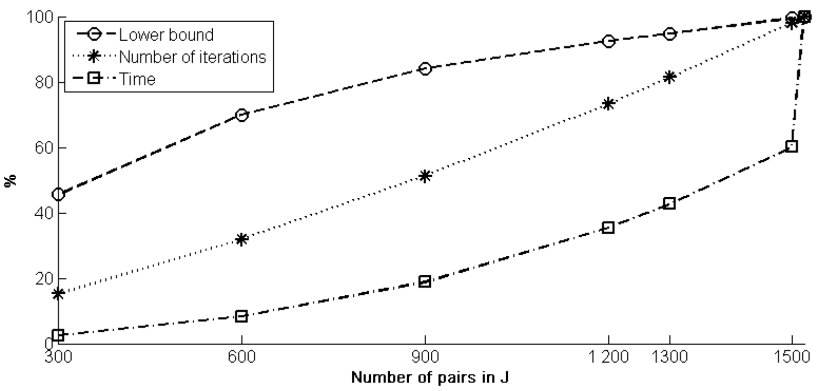


Figure 20 Increasing number of algorithm iterations depending on level of noise applied on image 23a

With constraints of higher arities, the algorithm runs rapidly slower. Instances containing 100 variables and 380 constraints (100 unary, 80 binary, 100 ternary and 100 quarternary) take approximately 40 minutes to solve. As expected, VJC found a higher lower bound in longer time. The slight increase in lower bound involved a 18% growth in time (Tab. 1).

	Lower bound	**Number of iterations**	**Running time**
VAC	1569	13883	2050
VJC	1589	15722	2428

Table 1 Results of lower bound in VAC and VJC algorithm.

7.2 Reviving Evaluation

We compared the algorithms for finding lower bound on WSCP with (VAC_r) and without (VAC) reusing information from the previous iteration. The two algorithms were tested on randomly generated binary problems. With increasing number of variables and constraints, the processing time of algorithm without reviving the values grew much faster although the average number of iterations does not differ. This was caused by increase in of average time of one iteration of VAC. The average time increased in every iteration, since VAC called more revise functions (Fig. 21).

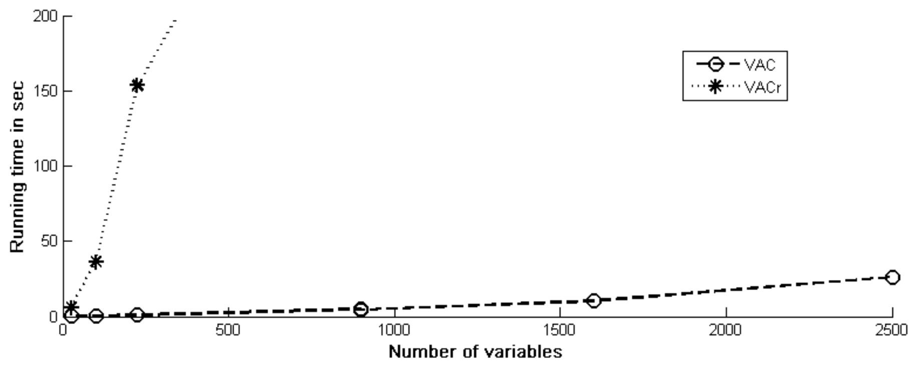


Figure 21 Execution time of VAC and VAC_r depends on number of variables.

Searching for the support turned out to be very expensive. It takes one third of processing time. Time complexity of the revise function was studied by Mohr and Henderson. They proposed an algorithm AC4 to improve the time complexity ([5]). AC4 stores informations during the revise function for the case that the same revise is recalled. They achieved $O(ed^2)$ time complexity, however, the space complexity increased to $O(ed^2)$.

In our algorithm, a tuple x_A is inconsistent if it the killer is set or $f_A(x_A) > \epsilon$. The information about killer is stored in constraints f_A such that $(A, B) \in J$. Hence when determining a tuple x_B to a conflict or a support, we have to prove all tuples x_A such that $(A, B) \in J$ consistent. Another approach would be to store the information directly in x_B. However, storing this information can be expensive as well and the information does not have to be used later. This topic is open for later research.

Extracting lines from image

Reviving of variables is tested on line extracting problem (see example 4.2).The complexity of the problem depends on the level of noise. In the range of $0 \ldots 1$, zero stands for the image without noise and one is represents a randomly generated gray scale image. The number of iterations depends on the level of noise (Fig 22).

An image (Fig. 23a) of a size 50×50 pixels contains 15 horizontal and 15 vertical lines. Different levels of noise are used to test the run of the algorithm reusing information from the previous iteration (Fig. 23b - Fig. 23e). The algorithm could recognize the images with noise smaller that 0.8. For 50×50, it took up to two minutes depending on the level of noise. The maximal size that the pseudocode was able to solve in few minutes was 100×100.

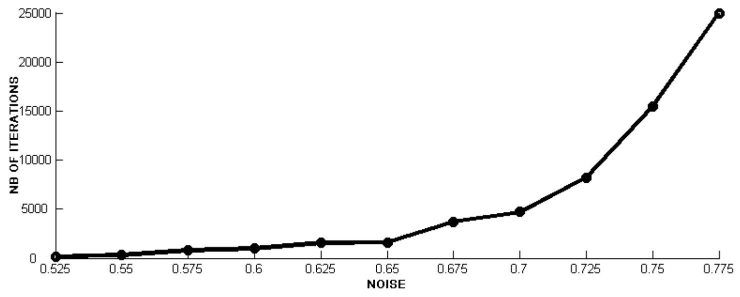
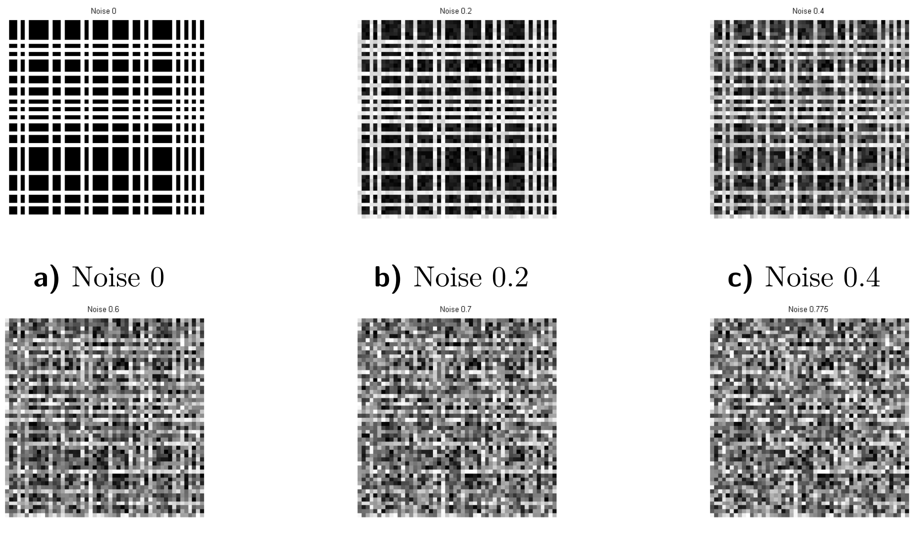


Figure 22 Increasing number of algorithm iterations depending on level of noise applied on image 23a



a) Noise 0 **b)** Noise 0.2 **c)** Noise 0.4

d) Noise 0.6 **e)** Noise 0.7 **f)** Noise 0.775

Figure 23 Gray scale noisy images of size 50×50 pixels containing 15 horizontal and 15 vertical lines

8 CONCLUSION

Constraint programming (CP) is widely applied in many areas. Constraint satisfaction problems (CSP) constitute the main class of CP problems, defined in a natural language of decision variables and constraints. CSP belong to the class of NP-complete problems, hence it can be problematic to find a solution for more complex problems. However, we can focus on improvement in solving particular parts of the problem. Weighted constraint satisfaction problem (WCSP) is a modification of CSP where hard constraints are replaced by soft constraint evaluating the variables.

In this thesis, we introduced VJC algorithm, based on VAC algorithm published in [3], which finds a lower bound on WCSP by applying equivalence-preserving transformations. The sequence of these transformations is carefully planned to lead to the increase in the lower bound. As results show, the VJC gives us freedom to choose between time and quality of the solution.

Furthermore, we have extended the VAC algorithm and so we have implemented a mechanism which remembers information reached in during the the algorithm. The extension leads to the decrease in time required by every iteration, because of less calls of revise function. The combination of VJC with reusing the informations from the previous iterations may be a topic for a later research.

Bibliography

[1] Javier Larrosa et al. “Solving weighted CSP by maintaining arc consistency”. In: *Artificial Intelligence* vol. 159.1-2 (2004), pp. 1–26.

[2] Martin Cooper and Thomas Schiex. “Arc consistency for soft constraints”. In: *Artificial Intelligence* vol. 154.1-2 (2004), pp. 199–227.

[3] M. C. Cooper et al. “Soft arc consistency revisited”. In: *Artificial Intelligence* 174.7-8 (May 2010), pp. 449–478. ISSN: 0004-3702.

[4] Alan K. Mackworth. “Consistency in networks of relations”. In: *Artificial Intelligence* vol. 8.issue 1 (1977), pp. 99–118.

[5] Roger Mohr et al. “Arc and path consistency revisited”. In: *Artificial Intelligence* vol. 28.issue 2 (1986), pp. 225–233.

[6] Eugene C. Freuder and Charles D. Elfe. “Neighborhood Inverse Consistency Preprocessing”. In: *In Proceedings of AAAI-96*. 1996, pp. 202–208.

[7] Eugene C. Freuder et al. “Synthesizing constraint expressions”. In: *Synthesizing constraint expressions* vol. 21.issue 11 (1978), pp. 31–47. ISSN: synthezising constraint expressions.

[8] Eugene C. Freuder. “A sufficient condition for backtrack-bounded search”. In: *Journal of the ACM* vol. 32.issue 4 (1985), pp. 755–761.

[9] Ugo Montanari. “Networks of constraints”. In: *Information Sciences* vol. 7 (1974), pp. 95–105.

[10] P. Janssen et al. “A filtering process for general constraint-satisfaction problems”. In: *[Proceedings 1989] IEEE International Workshop on Tools for Artificial Intelligence* (1989), pp. 420–427. URL: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=65349`.