Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Luděk Cigler

Constraint Satisfaction for HW/SW Verification

Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Doc. RNDr. Roman Barták, Ph.D.
Field of study: Theoretical Computer Science

2008

"All I really need to know I learned in kindergarten."
— Robert Fulghum

Acknowledgements

I thank my supervisor, Doc. RNDr. Roman Barták, Ph.D., for his patient guidance and for sharing his deep insight in the constraint satisfaction field with me. His critical comments have also helped me to make this thesis better.

I owe an immense debt to my parents, who supported me through the work on this thesis, as well as through my whole studies. This thesis is dedicated to them for their never-ending help.

Contents

Název práce: Použití programování s omezujícími podmínkami pro verifikaci HW/SW
Autor: Luděk Cigler
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce: Doc. RNDr. Roman Barták, Ph.D.
e-mail vedoucího: bartak@ktiml.mff.cuni.cz

Abstrakt: Programování s omezujícími podmínkami (CSP) je silným nástrojem pro modelování a řešení mnoha problémů v umělé inteligenci a operačním výzkumu. Verifikace HW a SW může využít CSP pro automatické vytváření testů. Hlavním požadavkem na CSP algorithmus (vzhledem ke generování testů) je rovnoměrné rozložení nalezených řešení. Studujeme několik stávajících algoritmů pro náhodné generování řešení klasických CSP problémů, a prezentujeme naše rozšíření těchto algoritmů na problémy s ohodnocenými podmínkami. Naše algoritmy testujeme na různých benchmarkových problémech.

Klíčová slova: programování s omezujícími podmínkami, náhodný výběr, generování testů

Title: Constraint Satisfaction for HW/SW Verification
Author: Luděk Cigler
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: Doc. RNDr. Roman Barták, Ph.D.
Supervisor's e-mail address: bartak@ktiml.mff.cuni.cz

Abstract: Constraint satisfaction techniques (CSP) are a powerful framework for modeling and solving various problems in artificial intelligence and operations research. Verification of HW and SW can profit from employing constraint satisfaction for test generation. The essential property of a CSP algorithm (wrt. test generation) is the uniform generation of solution samples. We present several algorithms for sampling solutions of a CSP and extend them so that they can be used for sampling solutions of CSP with preferences. We test the performance of our algorithms on various benchmark problems.

Keywords: constraint satisfaction, random sampling, test generation

Chapter 1

Introduction

Constraint satisfaction are a "versatilist" among artificial intelligence techniques. They can be used to solve a multitude of problems in both artificial intelligence and operations research, and quite often, they perform surprisingly well. Not only that: Their language of variables, domains and constraints allows its users to state the requirements on problem solutions in a simple declarative manner, almost completely eliminating the "cruft" of specifying *how* a problem should be solved. It's enough to say *what* needs to be solved. Eugene C. Freuder ([Fre97]) notes:

> *"Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."*

Verification, together with validation, is a part of a process used to ensure a product fits its desired purpose. In validation we ask: "Do we build the right product?", i.e. we check whether the product fulfils its desired purpose. On the other hand, in verification we ask: "Do we build the product right?", and we make sure the product behaves correctly according to some (more-or-less formal) specification.

When applied to hardware, we usually talk about *functional verification*, while in the software world, it is known as *program verification*. Proper verification is NP-hard in general – several techniques exist to tackle this problem for specific cases. *Formal verification* proves certain desired properties of an abstract model of the product. It offers real guarantee that the product behaves correctly, but can be quite hard in most cases. *Simulation* takes on an opposite approach – we generate a sufficient number of test cases which the product must pass in order to be declared "correctly behaving".

There have been several applications of constraint satisfaction techniques to verification of both hardware and software ([LFL$^+$95, DO91]), some of them have been applied by the industry as well (for microprocessor verification by IBM in [BESZ02] or for integrated circuit verification at Motorola in [YSP$^+$99]). The constraint satisfaction applications for software verification concentrate mostly on generating test data which the program is then executed on. McMinn ([McM04]) provides a nice survey of possible applications of constraint satisfaction to software test data generation. For hardware, test *programs* are generated which are executed either on a simulator (logical model) of the system, or on the real hardware itself.

A test must conform to a set of rules determined by the requirements of test engineers. A test of a software system may have to cover the whole public API. Tests of floating point unit may need to use instructions with positive and negative floating point numbers, together with special values. The test requirements may specify a number of instructions of a certain type in the test. Such requirements naturally fit into the range of problems constraint satisfaction is able to solve. It is therefore advantageous to use constraint satisfaction techniques to generate the tests automatically.

Executing a single test case may be very costly, so we would like to generate as few tests as necessary. On the other hand, any useful test suite should exercise a significant portion of the tested system, so the tests in the test suite should be as different from each other as possible. We therefore need to generate uniformly distributed random solutions to test-generation CSPs. This poses a problem to traditional constraint satisfaction techniques, namely those based on systematic search. These techniques usually employ some kind of heuristic, such as a special ordering of variables or values within a single domain, to speed up the search. This biases the resulting solutions.

The main goal of this work is to look into various ways of sampling solution space of a CSP and extend them so that they can be applied for automatic test generation. In chapter 3, we show that in general, the solution sampling task is #P-hard. This means that one cannot hope for an exact polynomial sampling algorithm for CSP. We then review three recent approaches to CSP sampling: Random walk strategies ([WES04]), which sample solutions to a boolean satisfiability problem (which every CSP can be transformed to); Bucket elimination methods ([DKBE02]); and sampling based on an algorithm for probabilistic inference in Bayesian networks called *Iterative Join-Graph Propagation* (also referred as IJGP, [GD06]).

Apart from specifying compulsory requirements on the test, test engineers have preferences on the resulting test – some code paths may be more preferred over another, it may be desirable to use as many floating point instructions as possible etc. The required distribution of solutions is not uniform anymore. Solutions which have higher preference should have higher probability than solutions which merely satisfy the hard constraints only.

To the best of our knowledge, there has not been any published work on the problem of sampling the solution space of a SoftCSP problem (CSP with preferences, or *soft constraints*). Our findings are presented in chapter 4. Two different approaches are presented: The first one relies on *Gibbs sampling*, a technique based on Monte Carlo Markov Chain theory. The second one extends work of Gogate and Dechter ([GD06]) on sampling hard constraints using IJGP.

The original IJGP sampling algorithm is very slow on problems with larger domains. We propose two extensions to it: First, we use constraint propagation methods to filter out values from the variable domains such that those values cannot yield any feasible solution. This helps to speed up the algorithm on problems with hard constraints. Second, we use an interval representation of the variable domains. This representation allows one to specify the number of intervals each domain can be separated to, and thus allows one to choose a trade-off between accuracy and speed of the algorithm. The intervals are heuristically adjusted so that the probability of finding a solution within an interval is approximately equal for all intervals in a single domain. The new algorithm which uses the interval representation, *Interval-IJGP sampling*, is up to an order of magnitude faster than the original IJGP-based algorithm on problems with domains of more than a handful values, without sacrificing much of the accuracy. The interval representation of domains is the main contribution of this work.

This thesis is organized as follows: In the following chapter, we introduce the reader to constraint satisfaction problems and its soft-constraint variants. We briefly refresh some probability theory terms, and introduce the theory of Monte Carlo Markov Chain algorithms. Chapters 3 and 4 discusses ways to sample from the solution space of a classical CSP and SoftCSP, respectively.

We have tested our algorithms for SoftCSP sampling on benchmarks from the *CostFunctionLib* benchmark library ([Sch]) and on a relaxed version of an *all-interval-series* problem proposed by Intel for the use in HW test generation. Results of our experiments are summarized in chapter 5.

Chapter 2

Preliminaries

2.1 Constraint Satisfaction Problems

Intuitively speaking, constraint satisfaction problems (CSPs) are described by *variables*, their *domains* (i.e. legal values of each variable) and *constraints* – rules which limit the values or combinations of values which the variables can be assigned. A solution to a CSP is simply an assignment of the variables which satisfies these rules.

The intuitive definition described above is formally captured by the notion of *constraint networks*. The definitions below are cited from [Dec03].

Definition 2.1.1. *A* constraint network $\mathcal{R}$ *is a triple* (X, D, C)*, where* $X = \{X_1, \ldots, X_n\}$ *is a finite set of* variables*,* $D = \{D_1, \ldots, D_n\}$ *is a set of respective* domains *and* $C = \{C_1, \ldots, C_t\}$ *is a set of* constraints*.*

A constraint C_i *is a relation* R_i *defined on a subset of variables* $S_i \subseteq X$*. The relation denotes allowed (satisfying) assignments of the variables.* S_i *is called a* scope *of* C_i*. Let* $S_i = \{X_{i_1}, \ldots, X_{i_r}\}$*, then* R_i *is a subset of a Cartesian product of respective variable domains, i.e.* $R_i \subseteq D_{i_1} \times \cdots \times D_{i_r}$*. We can thus view the constraint* C_i *as a pair* (S_i, R_i)*.*

To formally define what we described as a solution to a CSP, we first need to describe an *instantiation of variables.*

Definition 2.1.2. *An* instantiation *of a set of variables* $\{X_{i_1}, \ldots, X_{i_k}\}$ *is a tuple of ordered pairs* $((X_{i_1}, a_{i_1}), \ldots, (X_{i_k}, a_{i_k}))$ *such that no* X_i *appears twice in it. A pair* (X_j, a_j) *represents an assignment of value* a_j *from the domain of variable* X_j *to that variable. We can abbreviate the notation of* $((X_{i_1}, a_{i_1}), \ldots, (X_{i_k}, a_{i_k}))$ *to* $\overline{a} = (a_{i_1}, \ldots, a_{i_k})$*.*

An instantiation $\overline{a}$ satisfies a constraint $C_i = (S_i, R_i)$ iff it is defined over all variables in S_i and the combination of values of variables in S_i is present in the relation R_i.

Definition 2.1.3. *A* solution *of a constraint network $\mathcal{R} = (X, D, C)$, $X = \{X_1, \ldots, X_n\}$, is an instantiation of all its variables which satisfies all of the constraints. The set of all solutions is defined as*

$$sol(\mathcal{R}) = \{\overline{a} = (a_1, \ldots, a_n) \mid a_i \in D_i, \forall C_i \in C, C_i = (S_i, R_i) : \overline{a}[S_i] \in R_i\}$$

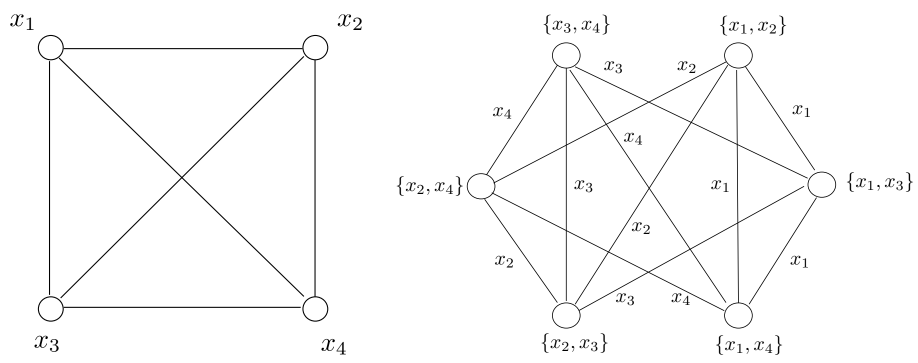
Here $\overline{a}[S_i]$ denotes the projection of an assignment $\overline{a}$ to the set of variables S_i.

The notion of constraint networks gives an emphasis on the relation between constraint satisfaction problems and graph theory – it owes its origin to the original applications of CSP. Indeed, graphs prove to be very useful in describing and reasoning about many CSP problems. The simplest representation of a constraint network as a graph is a *primal graph* – each node of such graph represents a variable, and an edge represents the fact that the two variables are sharing the same constraint, i.e. they are both members of the scope S_i of some constraint $C_i = (S_i, R_i)$. If there is no edge between two vertices (variables), no direct constraint between those exists.

A more straightforward transformation, albeit more complicated a structure, is a *constraint hypergraph*. There vertices represent variables again, but instead of edges we use *hyperedges* – scope of each constraint forms one such hyperedge. Dual to the hypergraph is the *dual constraint graph*, whose vertices represent scopes of the constraints, and each edge denotes that the two connected scopes share a variable. The edges are labeled by the shared variables.

Constraint Networks: An Example

To illustrate the use of constraint networks for modelling a CSP, we can describe the popular *N-Queens problem*. In this problem, our goal is to place n queens on the chessboard of size $n \times n$ so that no two queens share the same column, row or diagonal – i.e. the queens shall not threaten each other. Given $n = 4$, we have four variables $X = \{X_1, X_2, X_3, X_4\}$ to describe the rows in which each queen is placed (we assume implicitly that a queen X_i is placed in the i-th column). The domain of each of the variables is the same, $D_i = \{1, 2, 3, 4\}$. There are six binary constraints – one for each pair of variables from X. These constraints describe case-by-case the allowed



(a) Primal graph (b) Dual graph

Figure 2.1: Primal and dual graphs for the N-queens problem

positions for the pair, so that the two queens are “safe” from each other. Figure 2.1 shows primal and dual graphs of the N-queens problem.

2.1.1 Constraint propagation

There are two primary techniques for solving constraint satisfaction problem: *constraint propagation* and *search*. Constraint propagation is an inference technique that allows us to narrow the search space of possible assignments by removing single values or tuples of values which are guaranteed not to be present in any solution of the problem.

Consider two variables X, Y, both with the domain $\{1, 2, 3\}$. Propagation of a constraint $X < Y$ yields that $X < 3$ and $Y > 1$ – we can therefore remove 3 from the domain of X and 1 from the domain of Y. In general, constraint inference can find a complete solution to the problem. However, in most of the cases this is too hard, because it requires adding exponential number of new constraints. Therefore we restrict the inference to achieving some level of *consistency* for the problem.

Arc-consistency is a widely used consistency property, most importantly because several efficient algorithms exist to enforce it. When we speak about arc-consistency on problems with non-binary domains, we sometimes reffer to it as *generalized arc-consistency* (GAC). The formal definition is as follows:

Definition 2.1.4. *Let $\mathcal{R} = (X, D, C)$ be a constraint network, $c \in C$ be a constraint. Variable $X_i \in X$ is* arc-consistent *relative to c, if for every value $x_i \in D_i$ there exists a tuple t in the domain of variables in the scope* *of c such that t satisfies the constraints c and $t[X_i] = x_i$. A constraint is arc-consistent iff it's arc-consistent relative to each variable in its scope. The network $\mathcal{R}$ is arc-consistent iff all its constraints are arc-consistent.*

An efficient and simple-to-implement GAC-enforcing algorithm GAC-3 (algorithm 2.1) was proposed by Mackworth in [Mac77]. This algorithm views a constraint as a set of propagation methods $\mathcal{X} \to X_i$: Each method filters values from a set of variables $\mathcal{X}$ to the variable X_i by taking all values $x_i \in D_i$ and looking for a tuple in the domains of $\mathcal{X}$ that would satisfy the constraint. Mackworth has proven that the worst-time complexity of GAC-3 algorithm is $O(|C| \cdot r^2 \cdot d^{r+1})$, where $|C|$ is the number of constraints, d is the maximum domain size and r is the maximum arity of a constraint. On average, the algorithm usually performs much better.

Algorithm 2.1 GAC-3

```
Let $\mathcal{R} = (X, D, C)$ be a constraint network
$Q := \{\mathcal{X} \to Y | \mathcal{X} \to Y$ is a method for some constraint in $C\}$
**while** $Q$ is not empty **do**
  Select and delete $\mathcal{A} \to B$ from $Q$
  **if** the domain of $B$ changed after propagating $\mathcal{A}$ **then**
    **if** Domain of $B$ is empty **then**
      End with a failure; no solution exists
    **end if**
    $Q := Q \cup \{\mathcal{X} \to Y | \mathcal{X} \to Y$ is a method and $B \in \mathcal{X}\}$
  **end if**
**end while**
```

More efficient versions of GAC-enforcing algorithms have also been developed. Bessière and Régin present an algorithm *GAC-schema* with the time complexity $O(|C| \cdot d^r)$ in [BR97].

2.2 SoftCSP

Some constraint satisfaction problems may be too difficult (*constrained*) for a solution satisfying all the constraints to exist. Sometimes a solution may exist indeed, but we may need to express additional preferences apart from strict constraints. We can then look for solutions maximizing satisfaction of these preferences. All in all, it may be useful to consider some constraints to be "non-compulsory", i.e. *soft-constraints*.

Several formalisms have been developed to describe this sort of problems. Two of them are relevant to this work:

Valued CSP ([SFV95]) Every constraint has a cost, and we try to minimize the aggregated cost of unsatisfied constraints. Formally, we have a *cost structure* $(E, \otimes, >, \bot, \top)$. The set E is a set of costs totally ordered by a relation $>$. The binary operation $\otimes : E \times E \to E$ is associative and commutative. There exists an indifferent element $\bot$ for operation $\otimes$ $(\forall a \in E : a \otimes \bot = a)$ and an absorption element $\top$ for $\otimes$ $(\forall a \in E : a \otimes \top = \top)$. The operation $\otimes$ maintains monotonicity $(a > b \Rightarrow a \otimes c > b \otimes c)$.

Constraints C are mapped to the set of costs E by a function $\varphi : C \to E$. A solution of a valued CSP problem is such an assignment $\overline{a}$ of variables that the value of the aggregated cost function

$$v(\overline{a}) = \bigotimes_{\substack{c \in C \\ \overline{a} \text{ violates } c}} \varphi(c) \tag{2.1}$$

is minimal.

Semiring-based CSP ([BMR97]) Instead of constraints, we assign a fitness value to tuples. This denotes how well a constraint is satisfied by a particular tuple. A semiring is a structure $(P, \oplus, \times, 0, 1)$, where P is a set of preferences (e.g. real numbers), The binary operation $\oplus$ is commutative, associative and idempotent $(\forall a \in P : a \oplus a = a)$. It has a singular member 0 $(a \oplus 0 = a)$ and an absorption member 1 $(a \oplus 1 = 1)$. The binary operation $\times$ is also commutative and associative, with 1 as its singular member $(a \times 1 = a)$ and 0 as its zero member $(a \times 0 = 0)$.

Each constraint $c \in C$ has a fitness function δ_i which maps all tuples defined over variables in scope S_c to values in set of preferences P. A preference for a solution $\overline{a}$ is computed by aggregating the fitness functions for all constraints

$$\varrho(\overline{a}) = \mathop{\times}_{c \in C} \delta_c(\overline{a}[S_c]) \tag{2.2}$$

To compare the preferences, we need some ordering on P. The ordering can be defined using the additive operation $\oplus$: $a \leq b \iff a \oplus b = b$.

If $b \geq a$, we say that b is *better* than a. The goal is thus to find a solution A which maximizes its preference according to the ordering $\leq$ defined above.

Other major soft-constraint CSP formalism, *constraint hierarchies* (defined in [BFBW92]), classifies constraints into levels of preference, with the compulsory ones being in level *"required"*. The algorithms for solving constraint hierarchies are somewhat different from the ones used for the regular CSP or for one of the formalisms described above, and we will not discuss them any more.

The formalisms described above are in fact *meta-approaches* to constraints satisfaction with preferences. For modeling in practice, we usually use some of their instances which map some existing structure (e.g. real numbers $\mathbb{R}$) to the cost structure or semiring.

In our work, we are primarily concerned with *weighted CSP* problems ([SH81]). In weighted CSP, each constraint $c \in C$ defines a cost function g_c. This function maps tuples defined over variables in scope S_c to the set of non-negative real numbers $\mathbb{R}^+ \cup \{0\}$. The goal is to find such variable assignment $\overline{a}$ which minimizes the sum

$$v(\overline{a}) = \sum_{c \in C} g_c(\overline{a}[S_c]) \tag{2.3}$$

Weighted CSP can be viewed as a special case of semiring-based CSP: The semiring structure would be $(\mathbb{R}, \min, +, +\infty, 0)$ and for each constraint c, the semiring fitness function would be

$$\delta_c(\overline{a}) := -g_c(\overline{a}) \quad \forall \overline{a} \text{ assignment of variables in scope } S_c, \tag{2.4}$$

so that instead of minimizing the sum, we can maximize it. Note also that we can easily switch from minimizing the aggregated function to maximizing it and vice-versa.

Describing weighted CSP within the valued CSP meta-framework is a little tricky. Since there is no notion of *violated* constraint in the weighted CSP framework, we only define some weighted CSPs in terms of valued CSP framework.

In such problems, for every constraint c, the weighted CSP costraint function g_c should assign 0 to all tuples which satisfy the constraint. For all tuples that violate the constraint c, the function g_c should assign a fixed value β_c. Such weighted CSPs can be mapped to a valued CSP structure $(\mathbb{N} \cup \{+\infty\}, +, >, 0, +\infty)$, with constraint functions defined as

$$\varphi(c) = \max_{\overline{a} \text{ assignment of } S_c} g_c(\overline{a}) \tag{2.5}$$

2.3 Probabilistic Concepts

We'll briefly review some basic concepts from the probability theory, and introduce two representations of probability distributions – *Bayesian Networks* (BN) and *Markov Random Fields* (MRF). Unless otherwise stated, we use definitions from [DH03].

Definition 2.3.1. *Suppose that Ω is a non-empty set. A non-empty set $\mathcal{A}$ of subsets of Ω is a σ-*algebra*, if*

(i) $A \in \mathcal{A} \Rightarrow \overline{A} \in A$ *(here* $\overline{A} = \Omega \backslash A$*)*

(ii) $\forall n \in \mathbb{N} : A_n \in \mathcal{A} \Rightarrow \bigcup_{n=1}^{\infty} A_n \in \mathcal{A}$ *(here* $A_n \subseteq \Omega$*)*

The pair $(\Omega, \mathcal{A})$, where $\mathcal{A}$ is a σ-algebra of subsets of Ω, is called measurable space*.*

Definition 2.3.2. *Let $(\Omega, \mathcal{A})$ be a measurable space. A non-negative function $\mu : \mathcal{A} \to \mathbb{R}$ is called a* measure *iff:*

(i) $\mu(\emptyset) = 0$

(ii) $\forall i, j \in \mathbb{N}, i \neq j : A_i \cap A_j = \emptyset \Rightarrow \mu(\bigcup_{n=1}^{\infty} A_n) = \sum_{n=1}^{\infty} \mu(A_n)$

For a measurable space $(\Omega, \mathcal{A})$, we call the elements of the set Ω as *samples* and the elements of $\mathcal{A}$ as *events* (i.e. sets of samples).

Next we define the actual *probability function.* We will use the classical Kolmogorov axioms for its definition, though some of the properties are already present in the definition of measure.

Definition 2.3.3. *Let $(\Omega, \mathcal{A})$ be a measurable space. A* probability *function $\mathcal{P} : \mathcal{A} \to \langle 0, 1 \rangle$ is a measure on $\mathcal{A}$ such that:*

(i) $\mathcal{P}(A) \geq 0, A \in \mathcal{A}$

(ii) $\mathcal{P}(\Omega) = 1, \mathcal{P}(\emptyset) = 0$

(iii) *If* $\forall i, j \in \mathbb{N}, i \neq j : A_i \cap A_j = \emptyset$*, then* $\mathcal{P}(\bigcup_{n=1}^{\infty} A_n) = \sum_{n=1}^{\infty} \mathcal{P}(A_n)$

The triple $(\Omega, \mathcal{A}, \mathcal{P})$ *is called a* probability space.

Definition 2.3.4. *Let* $A, B \in \mathcal{A}, \mathcal{P}(B) > 0$*.* Conditional probability *of* A *depending on* B *is defined as*

$$\mathcal{P}(A|B) = \frac{\mathcal{P}(A \cap B)}{\mathcal{P}(B)} \tag{2.6}$$

Note that in the previous definition it is guaranteed that if $A, B \in \mathcal{A}$, then $A \cap B \in \mathcal{A}$. This is because $A \cap B = \Omega \setminus \left(\overline{A} \cup \overline{B}\right)$, and every σ-algebra is closed under complement and countable union of its members.

For $A, B \in \mathcal{A}$, we can denote the probability of their intersection $A \cap B$ as $\mathcal{P}(A, B)$.

Theorem 2.3.1. (Chain rule) *For any* $A_0, A_1, \ldots, A_n \in \mathcal{A}$ *such that* $\mathcal{P}(A_0, A_1, \ldots, A_n) > 0$ *it holds that*

$$\mathcal{P}(A_0, A_1, \ldots, A_n) = \mathcal{P}(A_0) \cdot \mathcal{P}(A_1|A_0) \ldots \mathcal{P}(A_n|A_0, A_1, \ldots, A_{n-1}) \tag{2.7}$$

Theorem 2.3.2. (Total probability) *Let* $\{B_n\}$ *be a countable sequence of disjoint events, let* $\mathcal{P}(\bigcup_n B_n) = 1$ *and* $\forall n \in \mathbb{N} : \mathcal{P}(B_n) > 0$*. For any* $A \in \mathcal{A}$ *then holds*

$$\mathcal{P}(A) = \sum_n \mathcal{P}(A|B_n) \cdot \mathcal{P}(B_n) \tag{2.8}$$

Theorem 2.3.3. (Bayes theorem) *With the same assumption as in previous theorem 2.3.2, together with the assumption* $\mathcal{P}(A) > 0$ *it holds that*

$$\mathcal{P}(B_m|A) = \frac{\mathcal{P}(A|B_m) \cdot \mathcal{P}(B_m)}{\sum_n \mathcal{P}(A|B_n) \cdot \mathcal{P}(B_n)} \tag{2.9}$$

Bayes theorem is particularly important in various probability inference algorithms, because it allows a transformation from one representation of conditional probability to the opposite.

Definition 2.3.5. (Independent events) *The events* $A, B \in \mathcal{A}$ *are* independent *iff* $\mathcal{P}(A, B) = \mathcal{P}(A) \cdot \mathcal{P}(B)$*.*

Theorem 2.3.4. *Let* $A, B \in \mathcal{A}$*. The following statements are equivalent:*

(i) *A and B are independent*

(ii) Let $\mathcal{P}(B) > 0$, then $\mathcal{P}(A|B) = \mathcal{P}(A)$.

(iii) Let $\mathcal{P}(A) > 0$, then $\mathcal{P}(B|A) = \mathcal{P}(B)$.

Definition 2.3.6. (Conditional independence) *Let $A, B, C \in \mathcal{A}$ and the joint probability $\mathcal{P}(B, C) > 0$. The event A is* conditionally independent *of B given C, iff*

$$\mathcal{P}(A|B, C) = \mathcal{P}(A|C) \tag{2.10}$$

In many situations we need some numerical representation of an outcome of a probabilistic experiment. If we throw a coin n times, the outcome in the language of probabilistic space $(\Omega, \mathcal{A})$ is a sequence of heads and tails of the length n. A numerical characteristic of this sequence may be the number of heads in it. To describe this characteristic formally, we introduce the concept of a *random variable*.

Definition 2.3.7. *For a given probability space $(\Omega, \mathcal{A}, \mathcal{P})$, a* random variable *is a real function $X : \Omega \to \mathbb{R}$.*

For a random variable X, we can ask questions such as: "How likely it is that the value of X is greater than 2?" For a given probability space $(\Omega, \mathcal{A}, \mathcal{P})$ this corresponds to the probability of an event $\{s \in \Omega : X(s) > 2\} \subseteq \mathcal{A}$. We often abbreviate it as $\mathcal{P}(X > 2)$ for short.

Definition 2.3.8. *Let X be a random variable. Its* probability distribution function *F_X is defined as*

$$F_X(x) = \mathcal{P}(X \leq x), \quad x \in \mathbb{R} \tag{2.11}$$

In this work, we are concerned only with *discrete probability distribution functions*. A probability distribution function F_X is discrete if there is a finite or countable sequence of real numbers $\{x_n\}, n \in \mathcal{N} \subseteq \mathbb{N}$ and corresponding positive real numbers $\{p_n\}$ such that $\sum_{n \in \mathcal{N}} p_n = 1$ and

$$F_X(x) = \textstyle\sum_{n \in \mathcal{N}, x_n \leq x} p_n, \quad \forall x \in \mathbb{R} \tag{2.12}$$

A corresponding random variable X is called a *discrete random variable*. For a discrete probability distribution function F_X, a *probability mass function* is a function that gives for every $x \in \mathbb{R}$ the probability that $X = x$. Sometimes we refer to a *domain* $D_X \subseteq \mathbb{R}$ of a discrete random variable X. This means that $\forall x \in \mathbb{R} \backslash D_X : P(X = x) = 0$ (the probability of values in D_X may be zero as well though).

For two random variables X, Y, a joint probability distribution is the distribution of X and Y together. In the discrete case, this can be defined as

$$\begin{aligned} \mathcal{P}(X = x, Y = y) &= \mathcal{P}(Y = y|X = x) \cdot \mathcal{P}(X = x) \\ &= \mathcal{P}(X = x|Y = y) \cdot \mathcal{P}(Y = y) \end{aligned} \tag{2.13}$$

For two probability distributions P, Q, we can measure their similarity by *Kullback-Leibler divergence* (referred also as *KL-divergence* or *relative entropy*). Typically, P corresponds to observations or measurements and Q corresponds to an ideal distribution. The KL-divergence for discrete distributions is defined as follows:

$$D_{KL}(P||Q) = \sum_i P(X = x_i) \cdot \log \frac{P(X = x_i)}{Q(X = x_i)} \tag{2.14}$$

KL-divergence is not a measure: in general, $D_{KL}(P||Q) \neq D_{KL}(Q||P)$. KL-divergence is always non-negative, and the lower the KL-divergence $D_{KL}(P||Q)$ is, the closer the distributions P and Q are. If $D_{KL}(P||Q) = 0$, the distributions are identical.

2.3.1 Bayesian Networks

Bayesian networks (or *belief*, *probabilistic* networks) are a graphical model to represent probability distributions by a set of variables and their probabilistic interdependencies. They were introduced by Judea Pearl in [Pea86]. To present them formally, let us mention the notion of a directed graph first.

Definition 2.3.9. *A* directed graph *is a pair $G = (V, E)$, where $V = \{X_1, \dots, X_n\}$ is a set of elements (vertices) and $E = \{(X_i, X_j) \mid X_i, X_j \in V,\ i \neq j\}$ is a set of edges. For each $X_i \in V$, we denote $pa(X_i) = \{X_j \mid (X_j, X_i) \in E\}$ (a set of its parents), and $ch(X_i) = \{X_j \mid (X_i, X_j) \in E\}$ (a set of its children). A directed graph is* acyclic *if it has no directed cycles.*

Definition 2.3.10. *Let $\mathcal{X} = \{X_1, \dots, X_n\}$ be a set of random variables over domains $D_1, \dots, D_n$. A* Bayesian network *is a pair (G, P) where $G = (\mathcal{X}, E)$ is a directed acyclic graph over the variables, and $\mathsf{P} = \{\mathcal{P}_i\}$, where $\mathcal{P}_i$ denotes conditional probability tables (called* CPTs*) $\mathcal{P}_i = [\mathcal{P}(X_i|pa(X_i))]$.*

In a Bayesian network, we assume that a random variable X is conditionally independent of any non-descendant variable Y given the values of

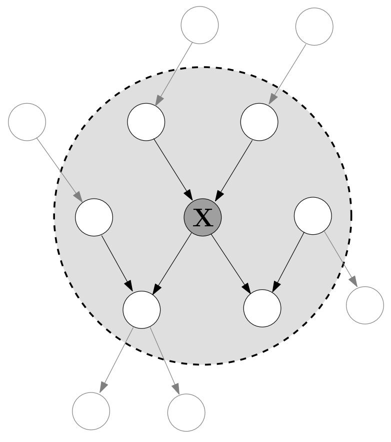


Figure 2.2: An example of Markov blanket. White nodes within grey dashed circle form a Markov blanket for random variable X.

the variable's parents $\mathrm{pa}(X)$, i.e. that $\mathcal{P}(X|\mathrm{pa}(X), Y) = \mathcal{P}(X|\mathrm{pa}(X))$. If we order the variables of the Bayesian network according to some topological ordering of the network graph (parents first), we can derive the *global semantics* of a Bayesian network from the conditional independence assumption and chain rule (theorem 2.3.1):

$$\mathcal{P}(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} \mathcal{P}(X_i|\mathrm{pa}(X_i)) \tag{2.15}$$

Another means of defining conditional independence in Bayesian network is by using the notion of *Markov blanket.* A Markov blanket of a variable X is a set of nodes ∂X containing the parents of X, children of X, and children's parents (see figure 2.2). A variable X is conditionally independent of any other random variable $Y \in \mathcal{X}$ given the values of all variables in the Markov blanket of X.

It can be easily shown that the space we need to represent a Bayesian network is (in some cases even exponentially) smaller than the space necessary to represent joint probability distribution (i.e. such that does not take independence into account). A CPT at some node X_i only needs to store one probability for every combination of values of parent nodes $\mathrm{pa}(X_i)$ with values of X_i itself. This means that for a Bayesian network where the maximum number of parents any node has is bounded by a constant, and the maximum domain size is also bounded by a constant, we only need linear space (with respect to the number of nodes in the network) to represent it. Had we represented the same probability distribution by a single table, we would need a single number for every combination of values of the variables. That would be clearly exponential (except for a trivial case of random variables with singleton domains).

We can further reduce the space needed to represent the probability distribution by using *canonical* probability distributions – i.e. distributions that are described by a set of parameters instead of a CPT; these include *Gaussian* (continuous) and *noisy-OR* (discrete) distributions. Recall that in order to specify a (univariate) Gaussian distribution, one only needs to store its mean μ and variance σ^2.

Bayesian networks can be used to sample from the solution space of a CSP. Consider the primal graph of a constraint network with some (arbitrary) ordering of variables as a special form of a Bayesian network. If we had a Bayesian network with CPTs set to represent probability distribution over CSP solution space, finding a solution, and even uniform sampling from that distribution would become trivial. We would only need to start in the root node, select its value according to the prior probability distribution specified by its CPT and continue recursively to its child nodes.

Of course, obtaining the correct values of CPTs is by no means an easy task. We can obtain full samples from the required probability distribution by other method (such as *Monte Carlo Markov Chain*, see section 2.4) and use them to approximate the correct CPTs. Bayesian networks also provide algorithms for learning CPTs from partially observable data such as Expectation-Maximization algorithm ([DLR77]). The problem is that such algorithm requires a method for complete inference over Bayesian network, i.e. computing the conditional probability $\mathcal{P}(X_i|\mathbf{e})$, where $\mathbf{e}$ is some evidence we have (an assignment of a subset of variables). This is NP-hard ([Coo90]), even for approximation bounded by some parameter ε. All known exact algorithms (such as *variable elimination*) have at least exponential time complexity depending on the width of the network graph. More advanced (and approximate) algorithms for probabilistic inference include Monte Carlo Markov Chain methods (in particular *Gibbs sampling*, [Pea87], which is also discussed in detail in section 4.2), *Mini-bucket Elimination* ([Dec96]) or *belief propagation* algorithms like *Iterative Join-Graph Propagation* ([DKM02], discussed in section 3.4).

2.3.2 Markov Random Fields

Another model for representing a joint probability distribution are *Markov Random Fields* ([KS80]).

Definition 2.3.11. *A* Markov random field (MRF) *over set of random variables $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$ with domains $D_1, D_2, \ldots, D_n$ is a pair (G, Φ), where $G = (V, E)$ is an undirected graph. Nodes in V represent random variables from $\mathcal{X}$, edges in E correspond to dependencies between random variables. The set $\Phi = \{\varphi_1, \varphi_2, \ldots, \varphi_m\}$ is a set of* potential functions. *A potential function φ_i has a domain of some (sub)clique k in G. It maps all possible assignments of variables in k to non-negative real numbers.*

In the following, we will use the term *node* and *variable* interchangeably, provided it does not yield confusion.

The joint probability distribution represented by MRF is defined as

$$P(X = \vec{x}) = \alpha \prod_{k} \varphi_k(\vec{x}_{\{k\}}) \tag{2.16}$$

Here X is the vector of all random variables in $\mathcal{X}$, $\vec{x}$ is a particular assignment of those variables, $\vec{x}_{\{k\}}$ is a restriction of assignment $\vec{x}$ to variables in clique k, and α is a normalizing constant which ensures that the total probability for all possible assignments adds up to 1.

A node $u \in V$ in MRF is conditionally independent from another node $v \in V$ given an assignment of some set of variables $S \subseteq \mathcal{X}$, iff every path between u and v goes through some node corresponding to a variable in S. A node is therefore conditionally independent of any other node given the values of its neighbours (this is an equivalent to Markov blanket defined for Bayesian networks).

The difference between BN and MRF is somewhat arbitrary – both can be used to describe the same probability distributions, and there are procedures to transform one model to another. Both also share similar limitations, namely the NP-completeness of probabilistic inference – the inference algorithms for MRF are similar to those used for BN. As we shall see in section 3.4, Markov random fields are better suited for describing probability distributions over solution spaces of CSP. They also allow an easier explanation of some inference algorithms, namely belief propagation.

2.4 Monte-Carlo Markov Chain

Markov chain is a sequence of random variables $\{X_0, X_1, X_2, \dots\}$, which follows the *Markov property* – the value of a variable X_{t+1} depends solely on the value of the variable X_t, i.e. $\mathcal{P}(X_{t+1}|X_0, X_1, \dots, X_t) = \mathcal{P}(X_{t+1}|X_t)$. The variables $X_0, X_1, \dots$ are called *states* of the Markov chain. The conditional probability $\mathcal{P}(.|.)$ is called the *transition kernel* of the chain. Markov chains can be discrete (random variables in the sequence can only have finite domains) or continuous, with infinite domain. Quite often the domain is a subset of $\mathbb{R}^d$.

Monte Carlo methods are general methods of estimating characteristics of a probability distribution (e.g. mean, variance, ...) by randomly sampling from that distribution. They are widely used in combination with Markov chains to form the so-called *Markov Chain Monte Carlo* methods (*MCMC*, [Gil95]). Monte Carlo estimation is a straightforward process: Given a probability distribution $\pi(X)$, sample independently n points $\{x_1, x_2, \dots, x_n\}$ from this distribution. Then compute an estimation of a given characteristics from these points, e.g. compute an arithmetic average of the samples to estimate the mean of $\pi(X)$.

It turns out that Markov chains can be used to represent a probability distribution and to sample from it. If we assume certain regularity conditions on the chain and iterate the chain long enough, the conditional probability $\mathcal{P}^{(t)}(X_t|X_0)$ (which describes how the state of t-th iteration depends on the initial state X_0) will gradually converge to a *stationary distribution*. Such distribution no longer depends neither on the initial choice of X_0 nor on the iteration index t.

The initial iteration phase is called a *burn-in* phase. After that, the state in which the Markov chain occurs is an approximately independent sample from the chain's stationary distribution.

In order to construct a Markov chain with a desired stationary distribution $\pi(.)$, *Metropolis-Hastings* algorithm can be used ([Has70]). The algorithm works as follows: First, it selects an initial state X_0; then in each iteration, a *candidate state* Y is generated from a so-called *proposal distribution* $q(.|X_t)$ (where X_t is the current state). The candidate is then accepted with probability

$$\alpha(X_t, Y) = \min\left(1, \frac{\pi(Y)q(X_t|Y)}{\pi(X_t)q(Y|X_t)}\right) \tag{2.17}$$

An interesting point is that we are completely free to choose the proposal distribution – the stationary distribution will always be $\pi(.)$. A popular stationary distribution is *Gibbs sampling* ([GG84], discussed in section 4.2).

The choice of a proposal distribution can have a strong impact on the rate of convergence towards the stationary distribution; even after the chain has converged, we may need many iterations to obtain proper estimates of the stationary distribution characteristics. The precise number of the required iterations depends on the *mixing time* of the chain – if the chain is *rapidly mixing*, the stationary distribution will be reached quickly (in less iterations). The *Jerrum-Sinclair theorem* ([SJ89]) gives an upper bound on the mixing time – it can be even exponential to the number of states of a Markov chain.

2.4.1 Simulated Annealing: An MCMC Method

Simulated annealing ([KGV83]) is a general meta-heuristic for searching global optimum of a given *fitness function* f. It serves also as a practical example of MCMC techniques.

Its basic concept is taken from metallurgy, and is as follows: The algorithm starts with a random initial solution of a problem. Then it proceeds iteratively: In each iteration a candidate neighbouring solution is selected randomly. For boolean formula satisfiability problem – *SAT* – this could be an assignment of boolean variables which differs in one variable only. The candidate solution is accepted with probability depending on the difference in fitness between the current and the candidate solution, and on a parameter T called *temperature*. The temperature is slowly decreasing during the run of the algorithm and the lower the temperature is, the less likely the algorithm is to accept a candidate solution with considerably worse fitness than the current solution. In order to prevent the algorithm from becoming stuck in a local optima, temperature may be increased after given number of iterations. This pushes the algorithm to explore other areas of the search space.

How does this process relate to MCMC? The key step of the transformation lies in the proper selection of a stationary distribution. Suppose that we want to minimize a fitness function f, then the stationary probability of state X is

$$\mathcal{P}(X) \propto \exp\left(-\frac{f(X)}{T}\right), \tag{2.18}$$

where T is the temperature in the current iteration and the $\propto$ symbol denotes direct proportionality. This means that when the temperature is high, the stationary distribution is “flattened” so that the algorithm can move easier around the search space, while with a low temperature, the distribution has “peaks” at low-fitness states and the algorithm will stay mostly in these states.

Chapter 3

Sampling CSP

The uniform probability distribution over the set of solutions to a CSP can be defined in an obvious way: Given a constraint network $\mathcal{R} = (X, D, C)$, the probability of a single variable assignment $\overline{a}$ is

$$\mathcal{P}(\overline{a}) = \begin{cases} \frac{1}{|sol(\mathcal{R})|} & \overline{a} \in sol(\mathcal{R}) \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

This equation suggests one obvious approach to obtain exact samples – use *brute force*. We could generate all solutions beforehand and sample from these stored solutions afterwards. This is of course impractical for all but the simplest problems.

On the other hand, naïve backtracking approach with random selection of variable values does not yield satisfactory results either. In [DKBE02], Dechter *et al.* present the following example: Define a constraint network over boolean variables $X = \{A, B, C, D, E\}$. The constraints are defined as implications: $\mathcal{C} = \{A \Rightarrow B, A \Rightarrow C, A \Rightarrow D, A \Rightarrow E\}$. If we would search over variables in an ordering A, B, C, D, E and select their values at random, selecting first value for $A := 0$ (with probability $\frac{1}{2}$) would allow all the other variables to take on both values from their domains – $\{0, 1\}$. Setting $A := 1$ (again with probability $\frac{1}{2}$) requires other variables to be equal to 1. The resulting probabilities for different assignments are then:

$$\mathcal{P}(\overline{a}) = \begin{cases} \frac{1}{2} & \overline{a} = (A = 1, B = 1, C = 1, D = 1, E = 1) \\ \frac{1}{32} & \overline{a}[A] = 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

This also shows that simply employing constraint propagation techniques does not help either – the problem was already arc-consistent at start.

3.1 CSP Sampling Complexity

To analyze the complexity of CSP sampling, let us borrow the notation and terms introduced in [Jer03]. A predicate $\pi : \Sigma^* \to \{0,1\}$, where Σ is some finite alphabet and Σ^* is the set of words defined over such alphabet, belongs to the complexity class NP iff there exists a *polynomial time witness-checking predicate* $\chi : \Sigma^* \times \Sigma^* \to \{0,1\}$ such that for all words $x \in \Sigma^*$

$$\pi(x) \Leftrightarrow \exists w \in \Sigma^* : \chi(x,w) \wedge |w| \leq p(|x|) \tag{3.3}$$

Informally speaking, there exists a polynomial time deterministic algorithm verifying for each input $x \in \Sigma^*$ and some potential "solution" w (with length bounded by a polynomial of the length of input x) that w is indeed a solution to input x.

Instead of asking whether a solution exists, we can ask how many solutions there are, i.e. we would like to have an algorithm for computing a counting function $f : \Sigma^* \to \mathbb{N}$. For class NP, an analogous class of counting problems is called #P. A counting function f is said to belong to #P, iff for a corresponding witness-checking predicate χ in NP it holds that

$$f(x) = |\{w \in \Sigma^* : \chi(x,w) \wedge |w| \leq p(|x|)\}| \tag{3.4}$$

A *randomised approximation scheme* for a counting problem f is a randomized algorithm[1] that for an input $x \in \Sigma^*$ and a threshold $\varepsilon > 0$ computes a number $N \in \mathbb{N}$ such that for every x the probability

$$\mathcal{P}(e^{-\varepsilon} f(x) \leq N \leq e^{\varepsilon} f(x)) \geq \frac{3}{4} \tag{3.5}$$

(the bound of $\frac{3}{4}$ is arbitrary; it is enough that it is greater than $\frac{1}{2}$).

A randomized approximation scheme is said to be *fully polynomial* (abbreviated as *FPRAS*) iff the algorithm runs in time polynomial of both $|x|$ and ε^{-1}.

It is not hard to see that deciding whether a CSP has a solution is NP-hard – just consider the well-known NP-complete problem of boolean formula satisfiability, SAT ([GJ79]), as a CSP instance. Computing the number of solutions to a CSP (a problem usually denoted as #CSP) is therefore #P-hard. Jerrum (in above cited [Jer03]) mentions the following:

[1]An algorithm which is allowed to make random choices during its computation

> *"If the decision version of a counting problem is NP-complete, then the counting problem itself cannot admit an existence of a FPRAS unless RP*[2] *= NP".*

This means that it's unlikely that a polynomial approximation algorithm would exist to compute the number of solutions to a CSP.

A polynomial sampling algorithm that would either output a solution sample or state that there is no solution for a given CSP would solve the decision version of the CSP problem, so its unlikely that such an algorithm would exist (unless RP = NP).

To further evaluate the complexity of sampling, let's define the notion of sampling problem properly: A *sampling problem* is specified by a relation $S \subseteq \Sigma^* \times \Sigma^*$ between problem instances x and "solutions" $w \in S(x)$, where $S(x) = \{w \in \Sigma^* | \chi(x, w) \wedge |w| \leq p(|x|)\}$.

An *almost uniform sampler* for a set of solutions $S \subseteq \Sigma^* \times \Sigma^*$ is a randomized algorithm that for an input $x \in \Sigma^*$ and a parameter $\delta > 0$ outputs a solution $W \in S(x)$ – in fact a random variable – such that the *variation distance* between the distribution of W and uniform distribution over $S(x)$ is at most δ. The variation distance for two probability distributions π, π' on a countable set Ω is defined as follows:

$$D(\pi, \pi') := \frac{1}{2} \sum_{\omega \in \Omega} |\pi(\omega) - \pi'(\omega)| \tag{3.6}$$

An almost uniform sampler is said to be *fully polynomial* (abbreaviated as *FPAUS*), if it runs in time polynomial with respect to $|x|$ and $\log \delta^{-1}$.

The key result in connecting samplers and approximation counting algorithms was proved by Jerrum, Valiant and Vazirani in [JVV86]. It states that for a fixed witness-checking predicate χ, corresponding counting problem $f : \Sigma^* \to \mathbb{N}$ and sampling problem $S \subseteq \Sigma^* \times \Sigma^*$, f admits a FPRAS iff S admits FPAUS.

This also means that since #CSP does not permit FPRAS (as noted above, and provided RP $\neq$ NP), the corresponding sampling problem does not permit FPAUS. It is unlikely that a polynomial sampling algorithm would exist that would produce samples from a distribution arbitrarily close to the correct distribution of an arbitrary CSP.

[2]A class of decision problems for which a polynomial randomized algorithm exists such that if the answer is NO, it answers NO, and if the answer is YES, it answers YES with a probability greater than $\frac{1}{2}$

3.2 Random Walk Sampling

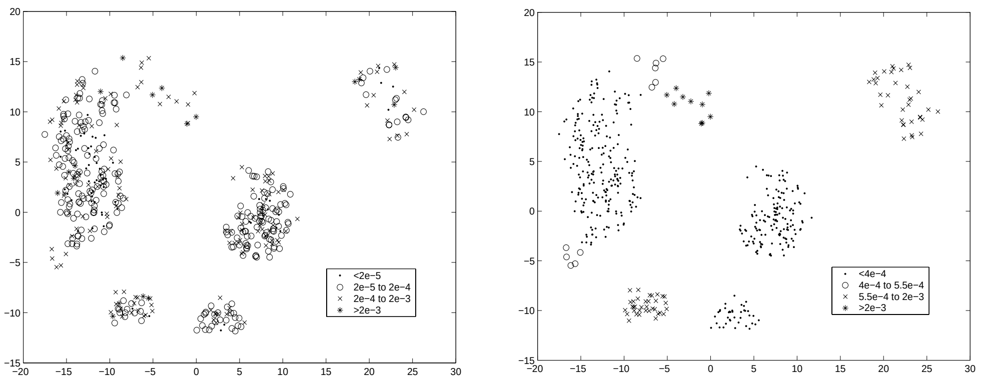
Wei, Erenrich and Seldman ([WES04]) use random walks to sample approximately uniformly from the set of solutions of SAT problems (instead of generic CSPs). Their results are particularly interesting for this work, as they use (apart from other data) boolean formulas generated for microprocessor verification ([Vel]).

As they state: *"MCMC methods are perhaps the best known methods for sampling from combinatorial spaces"*. Simulated annealing is one such method, as we have already discussed in section 2.4.1. It can be proved that simulated annealing with low temperature, provided that it reached a stationary distribution, samples uniformly from the state space of a special Markov chain. States of such Markov chain represent assignments of variables of a boolean formula. The desired stationary distribution assigns equal non-zero probabilities to all satisfying assignments, and zero probabilities to non-satisfying assignments (so that all the probabilities add up to 1).

The problem is that it takes up to an exponential number of steps (with respect to the number of clauses in the formula) for the simulated annealing algorithm to reach such stationary distribution, as discussed in section 2.4. The authors note: *"In principle, MCMC methods sample uniformly from states with an equal cost value (number of unsatisfied clauses)"*. However, to speed up finding a single solution (and thus a Markov chain state with zero unsatisfied clauses), random walks with a strong bias, such as *WalkSAT* ([SLM92]) are needed. As their experiments showed, WalkSAT is already not too bad in sampling uniformly from the solution space: For all formulas on which tests were performed, WalkSAT reached all satisfying assignments. For a synthetic boolean formula, the ratio between the number of visits for the most-often-visited and the least-often-visited formulas was in an order of 10^4, yet for a formula from microprocessor verification domain, this ratio decreased down to 7.

Their hybrid algorithm combines simulated annealing with WalkSAT: in each step, the algorithm chooses a WalkSAT move with a fixed probability p, and a simulated annealing move with a probability $1 - p$. Such algorithm was able to decrease the ratio between the most-often-visited and the least-often-visited solutions to 10 for the hardest formula, and to 4 for the formula from the microprocessor verification domain.

To understand the reasons why the hybrid algorithm worked better, it is useful to view the SAT solution spaces in 2-D using so-called *multidimen-*



(a) WalkSAT sampler (b) Hybrid sampler (WalkSAT + SA)

Figure 3.1: Solutions to SAT labeled by frequency of being found; Solutions are scaled down as 2-dimensional points

sional scaling (figure 3.1, cited from [WES04]). Solutions form relatively well-separated clusters.

When using the WalkSAT algorithm, solutions within one cluster are visited with different frequencies, while for the hybrid algorithm, they are all visited with a similar frequency. Before the hybrid algorithm found a solution, WalkSAT moves played more important role, as they lead the algorithm towards some cluster of solutions. After a solution was found, simulated annealing moves became more important, and the algorithm explored the entire cluster rather uniformly. Wei, Erenrich and Seldman have proved a theorem which states that *simulated annealing at zero temperature samples all solutions within a single cluster uniformly.*

3.3 Bucket Elimination Methods

Bucket elimination is a framework for expressing algorithms in a fashion similar to dynamic programming. It is not a single algorithm in a strict sense. Rather than that, it is an algorithmic *schema* which subsumes such algorithms as probabilistic inference algorithms for Bayesian networks ([Dec96]), constraint processing algorithms ([Dec97a]) and algorithms for automated reasoning ([Dec99]). More importantly, Dechter *et al.* ([DKBE02]) use bucket elimination as a basis for an algorithm to generate solutions to CSP uniformly.

The idea of bucket elimination can be easily explained on its specific instance, *directional resolution* ([DP60]), which is used for deciding the SAT problem. Assume that we are given a boolean formula in a conjunctive normal form, and that we have some ordering over variables used in that formula. For each variable, we create a *bucket*, which is a set of all clauses which contain that particular variable and do not contain any variable higher in the ordering. We then proceed from the highest-ordered variable and handle its bucket by resolution steps. As an example, one resolution step for clauses $a \vee b \vee c$ and $a \vee b \vee \neg c$ resolves them to

$$\frac{a \vee b \vee c \quad a \vee b \vee \neg c}{a \vee b} \tag{3.7}$$

Once a clause is resolved with some other clause, we place the resulting clause in the appropriate lower bucket. In general, we can use arbitrary functions over a finite number of variables instead of boolean clauses.

There are two major advantages in using a common framework such as bucket elimination for specifying an algorithm: clear (upper) bound on time- and space complexity is proven for the entire framework, and using common notation and terms can increase exchange of ideas from different disciplines. The complexity, both time- and space-, is exponential with respect to *induced width* of an *interaction graph* of the theory. For a constraint satisfaction problem, the interaction graph corresponds to the primal graph of the CSP.

Definition 3.3.1. *The* induced graph *of an ordered graph* (G, d) *is an ordered graph* (G^*, d) *where* G^* *is obtained from* G *by the following procedure: We process nodes from last to first along* d*. Once a node is processed, all of its parents are connected. The* induced width of an ordered graph (G, d) *is the width of its induced graph* (G^*, d)*. The* induced width of a graph*,* w^**, is a minimal induced width over all its orderings.*

The algorithm for generating random solutions to a CSP presented by Dechter *et al.* ([DKBE02]) is based on transforming a constraint network into a Bayesian network. Then, using known sampling methods for Bayesian networks ([Pea88]), one can generate solutions uniformly from the distribution defined by the Bayesian network. The transformation algorithm, *elim-count* (algorithm 3.1) uses bucket elimination with cost functions representing numbers of solutions which can extend a particular partial assignment of variables.

Figure 3.2 presents an example of an *elim-count* run on a CSP with three variables. There are two constraints in the CSP: $c_1 : A \neq B$ and $c_2 : A \neq C$.

Algorithm 3.1 Algorithm *elim-count*

```
Let $\mathcal{R} = (X, D, C)$ be a constraint network, $d$ an ordering of the variables
Partition $C$ into $bucket_1, \dots, bucket_n$ where $bucket_i$ contains all constraints whose largest variable is $X_i$.

for $p \in \{n, n-1, \dots, 1\}$ do
    $N^p = \sum_{X_p} \prod_{f_i \in bucket_p} f_i$
    Add $N^p$ to the bucket of largest variable in $\bigcup_{i=1}^{j} S_i - \{X_p\}$, where $j$ is the number of functions in $bucket_p$ and $S_i$ is the scope (set of variables) of the function $f_i$
end for

Return the number of solutions $N^1$ and the set of output buckets with computed and original functions
```

The set D_X denotes the domain of the variable X. The functions f_1, f_2 correspond to constraints c_1 and c_2, respectively. The function f_i is defined as follows:

$$f_i(x, y) = \begin{cases} 0 & x = y \\ 1 & x \neq y \end{cases} \tag{3.8}$$

The elim-count first computes the joint bucket function N^C:

$$N^C(a) = f_2(a, 0) + f_2(a, 1) \tag{3.9}$$

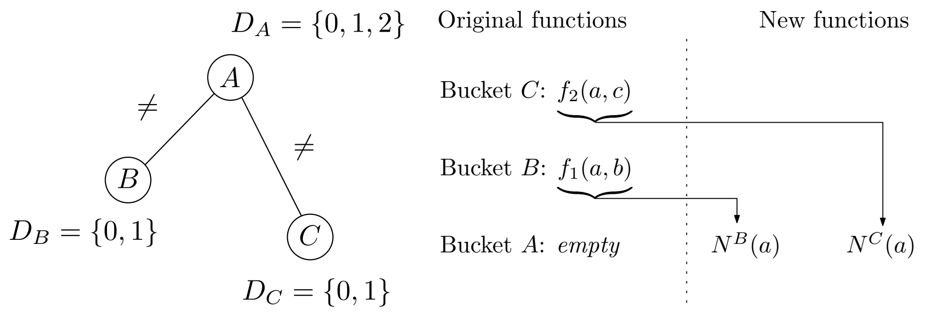
Then, it computes the joint bucket function for bucket B:

$$N^B(a) = f_1(a, 0) + f_1(a, 1) \tag{3.10}$$

Finally, it computes the joint bucket function N^A, which is also the total number of solutions to the example CSP:

$$\begin{aligned} N^A &= [f_1(0,0) + f_1(0,1)] \cdot [f_2(0,0) + f_2(0,1)] + \\ &\quad + [f_1(1,0) + f_1(1,1)] \cdot [f_2(1,0) + f_2(1,1)] + \\ &\quad + [f_1(2,0) + f_1(2,1)] \cdot [f_2(2,0) + f_2(2,1)] \\ &= 6 \end{aligned} \tag{3.11}$$

Using full bucket elimination on arbitrary constraint networks comes of course with the price of exponential complexity (as mentioned above), approximations are thus needed. Mini-bucket elimination ([Dec97b]) is a



(a) Primal graph of a CSP (b) *elim-count* run

Figure 3.2: Example of a bucket elimination algorithm *elim-count* with an ordering of variables A, B, C.

modified version of bucket elimination with adjustable levels of accuracy and efficiency. The main source of complexity for full bucket elimination is the processing of a single bucket – the time and space complexity of such processing is $O(exp(w))$ where w is the number of functions within that bucket. Mini-bucket elimination avoids this by partitioning each bucket into *mini-buckets* of size i (a parameter of the algorithm). These mini-buckets are then processed independently.

Dechter gives the following rationale for mini-bucket elimination: bucket elimination in its p-th step computes the function

$$N^p = \sum_{X_p} \prod_{i=1}^{j} f_i \tag{3.12}$$

Let $Q' = \{Q_1, \ldots, Q_t\}$ be a partitioning of $bucket_p$, where bucket Q_l contains functions $\{f_{l_1}, \ldots, f_{l_k}\}$. Then

$$N^p = \sum_{X_p} \prod_{l=1}^{t} \prod_{l_i} f_{l_i} \tag{3.13}$$

By moving the summation inside the first product, we obtain an upper approximation of N^p,

$$g^p = \prod_{l=1}^{t} \sum_{X_p} \prod_{l_i} f_{l_i} \tag{3.14}$$

Using operators other than summation, such as minimization or arithmetic mean, we can obtain other (lower, mean) approximations of N^p. However, no formal bounds on the quality of these approximations are given.

Dechter *et al.* ([DKBE02]) used small benchmark problems to measure the performance of mini-bucket elimination. They compared the Kullback-Leibler divergence between an exact uniform distribution (which they obtained by a brute force algorithm) with the distributions obtained for different parameter settings. The results showed a decrease of KL-divergence of an order of magnitude with increasing values of i (the mini-buckets size parameter), however the time also increased exponentially.

3.4 IJGP Sampling

Gogate and Dechter in [GD06] focused on a slightly more general approach to generate CSP solutions uniformly. Instead of counting solutions which extend a particular assignment of variables, they handle the uniform distribution as a probability distribution, approximate it, and sample from it using Monte-Carlo sampling.

The uniform distribution over CSP solutions can be expressed as follows: Given a constraint network $\mathcal{R} = (X, D, C)$, $C = \{C_1 = (S_1, R_1), \ldots, C_n = (S_n, R_n)\}$, the probability that a particular instantiation $\overline{a}$ is a solution is

$$\mathcal{P}(\overline{a}) = \alpha \prod_i f_i(\overline{a}[S_i]), \tag{3.15}$$

where i loops over all constraints, S_i is a scope of a constraint C_i, and the function f_i is defined as

$$f_i(\overline{a}[S_i]) = \begin{cases} 1 & \overline{a}[S_i] \text{ satisfies the constraint } R_i \\ 0 & \text{otherwise} \end{cases} \tag{3.16}$$

The normalization parameter $\alpha = 1/\sum_{\overline{a}} \prod_i f_i(\overline{a}[S_i])$ divides the product by the number of all solutions to the CSP. If we had some representation of such probability distribution, sampling would be then straightforward: Given an ordering of variables $X_1, X_2, \ldots, X_n$, sample $X_j = x_j$ from $\mathcal{P}(X_j | X_1, X_2, \ldots, X_{j-1})$ for each $j \in \{1, 2, \ldots, n\}$.

The probability distribution described above fits almost exactly into the scheme of Markov random fields, described in section 2.3.2. The primal graph of the CSP corresponds to the graph of a MRF, and the constraint functions f_i correspond to potential functions in MRF. This allows us to use existing inference algorithms for computing conditional probability distribution $\mathcal{P}(X_j | X_1, X_2, \ldots, X_{j-1})$ necessary for sampling. As we have already noted before, exact inference is known to be NP-hard.

3.4.1 Belief Propagation

Gogate and Dechter use *Iterative Join Graph Propagation* (*IJGP*, algorithm 3.2, [DKM02]) to approximate the probability distribution. This algorithm belongs to a class of *belief propagation* algorithms, introduced by Pearl in [Pea88] for Bayesian networks inference.

Yedidia *et al.* in [YFW03] give a nice survey of various other uses of belief propagation, as well as a thorough explanation of the algorithm. We will use their description of the basic belief propagation algorithm using *pairwise* Markov random fields – a MRF is pairwise if all cliques it contains have at most two nodes.

Suppose we have a MRF (G, Φ) and a partial assignment of variables $\mathbf{e}$ (evidence). The desired conditional probability $\mathcal{P}(X_i|\mathbf{e})$ can be expressed as

$$\mathcal{P}(X_i = x_i|\mathbf{e}) = \alpha\phi_i(x_i|\mathbf{e}) \prod_{j\in\mathcal{N}_i} m_{ji}(x_i|\mathbf{e}) \tag{3.17}$$

Here x_i is some value from the domain of X_i, $\mathcal{N}_i$ is the set of all neighbour nodes of X_i and α is a normalization constant ensuring that all probabilities add up to 1. The function m_{ji} is a *message* from the node of X_j to X_i. The function $\phi_i(x_i|\mathbf{e})$ is some local evidence in the node of X_i: If $(X_i, x'_i) \in \mathbf{e}$ for some value x'_i, then

$$\phi_i(x_i|\mathbf{e}) = \begin{cases} 1 & \text{if } x_i = x'_i \\ 0 & \text{otherwise} \end{cases} \tag{3.18}$$

Otherwise, $\phi_i(x_i)$ is a product of functions from Φ which are defined on the variable X_i only.

Informally speaking, a message describes how likely the node X_i has the value x_i from the point of view of X_j. Formally, we obtain a message by the following update rule:

$$m_{ji}(x_i|\mathbf{e}) \leftarrow \sum_{x_j\in D_j} \phi(x_j|\mathbf{e})\psi_{ji}(x_i, x_j|\mathbf{e}) \prod_{k\in\mathcal{N}_j\setminus\{i\}} m_{kj}(x_j|\mathbf{e}) \tag{3.19}$$

Here we are summing over the whole domain D_j of variable X_j. The function $\psi_{ji}(x_i, x_j|\mathbf{e})$ is a product of all functions from Φ which are defined over variables X_i, X_j (here again it also depends on the evidence in a similar sense to ϕ_i). A message m_{ji} thus depends on all previous messages to the node X_j *except* the message going "backwards" (m_{ij}).

The rationale behind these rules can be illustrated on a simple example of a tree-like MRF, i.e. an MRF which does not contain any loops (in this

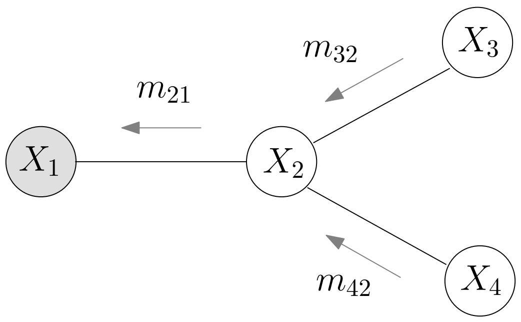


Figure 3.3: An example of a tree-like Markov random field. The grey node is the one on which we are performing inference; the arrows show the flow of the messages in belief propagation

case, the algorithm corresponds to exact inference). Consider the MRF shown on figure 3.3, and an empty evidence **e**. We would like to compute the probability distribution over X_1. As described above, this is defined as

$$\mathcal{P}(X_1 = x_1) = \alpha \phi_1(x_1) m_{21}(x_1) \tag{3.20}$$

Using the message update rule, we substitute the message m_{21} such that

$$\mathcal{P}(X_1 = x_1) = \alpha \phi_1(x_1) \sum_{x_2 \in D_2} \phi(x_2) \psi_{21}(x_1, x_2) m_{32}(x_2) m_{42}(x_2) \tag{3.21}$$

Finally, by expanding the messages m_{32} and m_{42} again with the message update rule, we get

$$\begin{aligned} \mathcal{P}(X_1 = x_1) \quad = \quad & \alpha \phi_1(x_1) \textstyle\sum_{x_2 \in D_2} \phi(x_2) \psi_{21}(x_1, x_2) \cdot \\ & \cdot \textstyle\sum_{x_3 \in D_3} \phi(x_3) \psi_{32}(x_2, x_3) \cdot \\ & \cdot \textstyle\sum_{x_4 \in D_4} \phi(x_4) \psi_{42}(x_2, x_4) \end{aligned} \tag{3.22}$$

By reorganizing the sums, we can see that this equation is equivalent to an exact inference equation for tree-like MRFs:

$$\mathcal{P}(X_1 = x_1) = \alpha \sum_{x_2, x_3, x_4} \prod_k \varphi_k(\vec{x}_{\{k\}}) \tag{3.23}$$

where $\vec{x} = (x_1, x_2, x_3, x_4)$.

Surprisingly, belief propagation can be also applied to some MRFs which *do* contain loops. In this case, we need to iterate the belief update process:

We start with some arbitrary messages (e.g. uniform ones) and iteratively update them using the rules described above. There are cases in which such approach fails ([MWJ99]), but there are also cases in which it performs well ([Pea88]).

Sometimes it is even advantageous to group several variables into one cluster and perform an inference over such *cluster-graph*. This is the basic idea of a *Generalized belief propagation* algorithm, described by Yedidia, Freeman and Weiss in [YFW00].

3.4.2 IJGP: An Algorithm

The IJGP algorithm is an example of a generalized belief propagation algorithm. It transforms a *factored* probability distribution of a constraint network into a special structure called *join-graph* and then performs iterative message passing over this graph.

Definition 3.4.1. *Given constraint network $\mathcal{R} = (X, D, C)$ and a factored distribution $\mathcal{P}(\overline{a}) = \prod_i C_i(S_i)$, $S_i \subseteq X$ and $C_i \in C$ are factors of the distribution, a* join-graph *for $\mathcal{R}$ is a triple $JG = (G, \chi, \psi)$, where $G = (V, E)$ is a graph and χ and ψ are labeling functions. These functions associate with each vertex $u \in V$ two sets: Variable label $\chi(u) \subseteq X$, and constraint label $\psi(u) \subseteq C$, with the following properties:*

(i) For each $C_i \in C$, there is exactly one vertex $u \in V$ such that $C_i \in psi(u)$. For such u it must also hold that $S_i \subseteq \chi(u)$.

(ii) (Connectedness property) *For each variable $X_i \in X$, the set $\{u \in V | X_i \in \chi(u)\}$ induces connected subgraph of G.*

Nodes of a join graph are therefore clusters of variables (one variable can be in multiple clusters) and constraints (a constraint has to be only in one cluster). In an *arc-labeled join-graph*, edge (u, v) is labeled by a labeling function $\theta(u, v) \subseteq \chi(u) \cap \chi(v)$. In such a join-graph it must hold that for each variable $X_i \in X$, the set of nodes containing X_i must induce a subgraph connected by edges whose label $\theta(e)$ contains X_i.

The IJGP algorithm is then used to compute the posterior probability $\mathcal{P}(X_j|\mathbf{e})$. It works iteratively – in each iteration it passes *messages* over each edge in the join graph (in both directions). Here a message $m_{(u,v)}$ is defined over variables in $\theta((u, v))$, and is computed in the same way as in an ordinary belief propagation algorithm.

Algorithm 3.2 Iterative Join-Graph Propagation

```
Let $\mathcal{P}(\overline{a}) = \prod_i f_i(S_i)$ be a factored probability distribution for constraint network $\mathcal{R} = (X, D, C)$
Let $JG = (G = (V, E), \chi, \psi)$ be a join graph for distribution $\mathcal{P}(\overline{a})$
Let $\mathbf{e}$ be an initial evidence of variables $I \subseteq X$

Let $m_{(u,v)}$ be a *message* from vertex $u \in V$ to $v \in V$
$cluster(u) := \chi(u) \cup \{m_{(u,v)} | (v, u) \in E\}$
$cluster_v(u) := cluster(u) \backslash \{m_{(v,u)}\}$
$elim(u, v) := \chi(u) \backslash (\theta((u, v)))$
**repeat**
  **for** each vertex $u \in V$ along some ordering and back **do**
    Assign the evidence from $\mathbf{e}$, set $\chi(u) := \chi(u) \backslash I$
    **for** each edge $(u, v) \in E$ **do**
      Compute message: $m_{(u,v)} := \alpha \sum_{elim(u,v)} \prod_{f \in cluster_v(u)} f$
    **end for**
  **end for**
**until** maximum number of iterations is exceeded or Kullback-Leibler divergence between old and new messages is less than some threshold
```

The output of IJGP can be used to compute the posterior probability $\mathcal{P}(X_j|\mathbf{e})$ for partial assignment $\mathbf{e}$ of variables $I \subseteq X$ and $X_j \in X \backslash I$: Select a node u of the join graph JG where $X_j \in \chi(u)$ and use the following equation:

$$\mathcal{P}(X_j|\mathbf{e}) = \alpha \sum_{\chi(u) \backslash \{X_j\}} \prod_{f \in cluster(u)} f \tag{3.24}$$

Here again we sum over products of constraint functions and incoming messages for all combinations of values of variables in $\chi(u) \backslash \{X_j\}$.

The complexity of the algorithm depends exponentially on the maximum size of a single cluster. The following theorem proven by Larrosa *et al.* in [LKD01] states it precisely:

Theorem 3.4.1. *The time complexity of one iteration of IJGP is $O(g \cdot (n + N) \cdot d^{w^*+1})$, where g is the maximum degree of a node in the join graph, n is the number of variables, N is the number of nodes in the join graph, d is the maximum domain size and w^* is the maximum number of variables in some cluster. The space complexity of IJGP is $O(N \cdot d^{\theta})$, where θ is the maximum size of an edge label (separator).*

Algorithm 3.3 Schematic Mini-Bucket Elimination

```
Let $i$ be the maximum desired size of a mini-bucket

Order variables $X = \{X_1, \ldots, X_n\}$ heuristically to minimize induced
width of the underlying primal constraint graph
Create a bucket for each variable in $X$
Place the scope $S_i$ of each constraint function $f_i$ in a bucket of the variable
from that scope which is the last in the computed ordering

for $j \in \{n, \ldots, 1\}$ do
  Heuristically partition the scopes in $bucket_j$ into mini-buckets having
  at most $i$ variables
  For each created mini-bucket $M$, create a new scope $S := M \backslash \{X_j\}$ and
  place it in the bucket of the last variable in $S$.
  When the mini-bucket $M'$ is created which contains $S$ (in the future),
  create an arc between $M'$ and $M$.
end for
```

3.4.3 Constructing a Join-Graph

In our discussion of the IJGP algorithm, we have skipped an important step: How does one construct a join-graph for a given constraint network and factored probability distribution? Moreover, since the time complexity of IJGP is bounded exponentially by the maximum size of a join-graph cluster w^* (as we have seen above), we would like to construct a join-graph whose cluster size would be bounded by some parameter i.

Dechter *et al.* in [DKM02] propose an algorithm called *join-graph structuring* (algorithm 3.4) to do that. First, it splits the variables into mini-buckets using a version of mini-bucket elimination (which was discussed also in section 3.3). This version works with function scopes (or "schemas") only, hence the name "schematic mini-bucket". It is described in algorithm 3.3. Function scopes in a single bucket are partitioned into mini-buckets of size i. This partitioning is done heuristically. If, however, a function has scope larger than i, a new separate mini-bucket for the scope of this function is created. This also means that for constraints with large scopes (including global constraints), restricting the mini-bucket size to i does not help.

After the mini-buckets are created, we create labeled edges between them – either labeled by the separator of the two mini-buckets or labeled by the

Algorithm 3.4 Join-Graph Structuring

```
Let $i$ be the maximum desired size of a mini-bucket

Perform *schematic mini-bucket elimination* (algorithm 3.3) with parameter $i$.
For each created mini-bucket $M$ create a node $u$ in the join-graph. The variable label is $\chi(u) := M$.
Add the arcs $(M, M')$ created by schematic mini-bucket elimination to the join-graph as $(u, u')$ and label them with the node separator $sep(u, u') = \chi(u) \cap \chi(u')$
Add arcs between nodes corresponding to mini-buckets from the same bucket, and label them with that bucket variable
```

variable of the original bucket. Figure 3.4 shows an example of an original constraint graph and created join-graph.

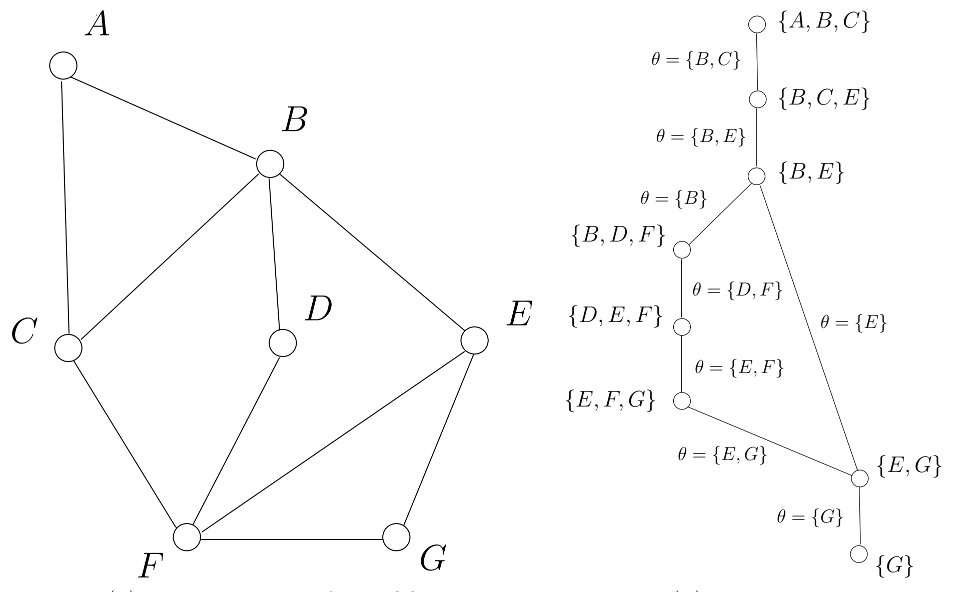
3.4.4 Sampling with IJGP

In their CSP sampling algorithm, Gogate and Dechter in [GD06] run IJGP prior to sampling, to obtain initial estimates of CSP probability distribution. Then, after every K variables have been instantiated, they re-run IJGP with the current partial sample as evidence (K is a parameter of the algorithm). As the computed distribution is only an approximation of the true distribution, it needs to be updated when a sample which is not a CSP solution is generated (such sample is rejected). To avoid starting the sampling anew when such sample is generated, Gogate and Dechter suggest using classical CSP techniques such as backtracking, backjumping and no-good learning.

The experiments presented by Gogate and Dechter were performed on problems with binary domains, and for such problems, the algorithm was competitive to WalkSAT-based algorithms. We discuss the performance aspects of running IJGP on larger domains in chapter 4.

3.5 Sampling CSP in Practice

All approaches described above have concentrated itself on problems with binary domains. The practice of HW/SW test generation however requires handling domains much larger than in any conventional CSP – sometimes



(a) Primal graph of the CSP (b) Join-graph

Figure 3.4: Example of primal constraint graph and corresponding join-graph with maximum mini-bucket size 3 and an alphabetical ordering of variables

their size is up to 2^{64}. Bin *et al.* ([BESZ02]) use three different structures to represent such domains – union of intervals, union of binary masks and binary decision diagrams ([Ake78]). They sample from these domains by simply randomly selecting a value for a given variable – an approach we have already shown has poor theoretical performance. They even allow the use of variable ordering heuristics in order to speed up the search for solutions.

A rather different approach to random sampling is used in [NSZ07], although it is not a CSP technique in a strict sense: The authors model a HW test case generation problem as a linear programming (LP) problem (which could be viewed as a subset of CSP, though the techniques for solving LP are quite different). Then they randomly select variables which are assigned a fixed value. This leads to a distribution of samples which is uniform enough for their examples.

Both the results described above do not struggle to achieve a uniform distribution of samples. This suggests that in practice, it is usually enough to generate samples quite different from each other.

Chapter 4

Sampling SoftCSP

4.1 Probability Distribution for SoftCSP

A probability distribution over SoftCSP problems can be defined in several ways: We could sample only *optimal* solutions of the problem, and among several optimal solutions, we can choose at random. Or we may sample all of the *feasible* solutions (i.e. those that satisfy all the hard constraints) with a probability proportional to their fitness function.

Sampling only the optimal solutions can be achieved by using the techniques described in the previous chapter. First, we find the cost of an optimal solution by some optimization algorithm. Then we add a new constraint which requires the cost of a feasible solution to be equal to the cost of an optimal solution.

We have chosen the second approach (sampling proportionally from the set of all feasible solutions), because from our point of view it is more generic: It allows us to parametrize the resulting distribution based on the needs of a specific problem.

The probability distribution over SoftCSP can be described similarly to the one used for hard-constraints in section 3.4:

$$\mathcal{P}(\overline{a}) = \alpha \prod_{c \in C} f_c(\overline{a}[S_c]) \tag{4.1}$$

For hard constraints, the constraint functions f_c are the same as in section 3.4. For soft constraint c, we define the constraint function as

$$f_c(\overline{a}[S_c]) = \sigma^{g_c(\overline{a}[S_c])} \tag{4.2}$$

where g_c is the constraint cost (or fitness) function (defined in section 2.2), and $\sigma > 0$ is a parameter. If we want to maximize overall fitness, we set $\sigma > 1$. On the other hand, to minimize the fitness (or cost), we should set $0 < \sigma < 1$.

Note that in the equation 4.1, we could replace the product of all soft-constraint functions with a single function $\sigma^{\sum_c g_c(\overline{a}[S_c])}$. This means that sample probability grows exponentially with the sample fitness.

4.2 Gibbs Sampling

Gibbs sampling ([GG84]) is one of the most widely used *proposal distributions* for Metropolis-Hastings algorithm (see section 2.4 for the discussion about what a proposal distribution is). Instead of generating candidate states of Markov chain at once, it divides a single state X (which is usually a vector, at least in finite case) into components $\{X_1, X_2, \ldots, X_h\}$ and generates these components separately (the components can be vectors as well, of possibly varying dimension). In Gibbs sampling, candidate states are always accepted. The i-th component of the candidate state Y_i is generated from the following distribution:

$$q_i(Y_i|X_1, X_2, \ldots, X_h) = \pi(Y_i|X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_h), \tag{4.3}$$

where $\pi(.|.)$ is a conditional distribution derived from the required stationary distribution of the Markov chain $\pi(.)$.

This may seem contradictory to the initial purpose of avoiding to generate samples directly from the probability distribution $\pi(.)$. However, in many applications (such as in Bayesian networks) it is easier to determine the conditional distribution $\pi(.|.)$ than to determine the joint distribution $\pi(.)$.

We use Gibbs sampling with the SoftCSP probability distribution described above as the stationary distribution of the Markov chain. At the beginning, the sample is initialized randomly. The components of the sample are the CSP variables itself. After that, we proceed iteratively: In each step we update all values for all CSP variables – we pick a variable X_j to update, compute the distribution of $\mathcal{P}(X_j|X_1, \ldots, X_{j-1}, X_{j+1}, \ldots, X_n)$ and sample from it. See algorithm 4.1 for a pseudo-code of the algorithm.

Computing the conditional probability distribution is easy: Since we have fixed values for all other variables except X_j, we iterate over all values in the domain of X_j to compute the joint probability by equation 4.1.

Algorithm 4.1 Gibbs sampling

```
Let $\mathcal{R} = (X, D, C)$ be a constraint network
Initialize the sample $\overline{a}$ randomly
repeat
  for each variable $X_j \in X$ do
    Let $C_{X_j} := \{c \in C | X_j \in S_c\}$
    {Compute the distribution $\mathcal{P}(X_j|X_1, \ldots, X_{j-1}, X_{j+1}, \ldots, X_n)$}
    for each value $x_i \in D_j$ do
      $\overline{a}' := \overline{a}$, $\overline{a}'[X_j] := x_i$
      $p_i := \prod_{c \in C_{X_j}} f_c(\overline{a}'[S_c])$
    end for
    Let $\vec{p} := (p_1, p_2, \ldots, p_{|D_j|})$
    Normalize the vector $\vec{p}$
    Sample a new value $x_j$ of variable $X_j$ from the distribution defined
    by the vector $\vec{p}$ and assign it to the current sample $\overline{a}$
  end for
  if we have exceeded the number of burn-in iterations then
    Output the current sample $\overline{a}$
  end if
until we have generated a given number of samples
```

Before we output the first sample, we need to perform a sufficient number of iterations so that the Markov chain "forgets" its initial state (a *burn-in* phase). There are several "rules of thumb" on determining the length of the burn-in, some of them are discussed in [Gil95].

In general, we should observe several statistical characteristics of the samples the algorithm outputs (such as sample mean, variance, or KL-divergence with respect to some target distribution). When these characteristics converge, we can quit the burn-in phase.

4.3 IJGP + GAC Sampling

Iterative join-graph propagation can be used for sampling from the probability distribution of a SoftCSP problem without any changes. However, our preliminary testing showed that such an algorithm had a hard time satisfying hard constraints, and its run-time performance on problems with larger domains was very slow.

Algorithm 4.2 IJGP + GAC sampling

```
Recursively sample from all variables $\{X_1, \ldots, X_n\}$

if we have a full sample then
  Output the sample and return success
else
  Propagate constraints using GAC-3
  if some domain is empty then
    Return failure
  else
    Let $X_i$ be the current variable to be sampled
    Let $\mathbf{e}$ be the current partial sample (evidence)
    Every $K$-th recursion step, run IJGP to compute $\mathcal{P}(X_i|\mathbf{e})$
    while there are some values from the domain of $X_i$ to try do
      Sample value $v$ from $\mathcal{P}(X_i|\mathbf{e})$
      Add $(X_i, v)$ to the evidence $\mathbf{e}$
      Recursively call sampling procedure for variable $X_{i+1}$
      if the previous recursive call succeeded then
        Return success
      else
        Remove $v$ from the probability distribution $\mathcal{P}(X_i|\mathbf{e})$
      end if
    end while
    Return failure
  end if
end if
```

To address these shortcomings, we decided to exploit the power of constraint inference by incorporating *generalized arc-consistency* enforcing into the algorithm (see section 2.1.1 for its definition).

We maintain generalized arc-consistency during sampling: We run GAC-3 propagation algorithm prior to sampling, and then we run it every time some value has been selected for a next variable. This avoids unnecessary computations with values which yield no solution in the IJGP message updating step. The sampling algorithm is described in algorithm 4.2.

Why can we use GAC during IJGP sampling? Recall the message updating phase of IJGP: For a join-graph node $u \in JG$, we summed over all combinations of possible values for the variables in its variable label $\chi(u)$. Now let $x_i \in D_i$ be the value for a variable $X_i \in \chi(u)$ which was filtered out during GAC-3. Suppose that the constraint $c \in C$ responsible for propagating that value out is indeed a member of the constraint label of the node u, $\psi(u)$. Any combination of values containing the value x_i would have zero probability because of the constraint function f_c. We needn't have bothered with that combination at all.

What about the case where the constraint c which filtered x_i out is not present in the constraint label $\psi(u)$? Here we can use the connectedness property of the join-graph: The set $V_i = \{v \in V | X_i \in \chi(v)\}$ induces a connected subgraph, which edge labels contain the variable X_i. It must hold that $c \in \psi(u')$ for some node $u' \in V_i$. This means that sooner or later, a message would propagate from u' to u such that it would carry zero probability for x_i. After that, we don't need to bother with the value x_i any more.

Apart from adding the GAC-enforcing, our sampling algorithm is the same as the one described by Gogate and Dechter in [GD06]. The sampling algorithm contains a form backtracking in it. This means that in the worst case, it would need to explore the whole solution space to find a sample, and the worst-case time complexity is exponential with respect to the number of CSP variables. The major operations in one recursion step are the GAC-3 and IJGP. The worst-case time complexity of GAC-3 is $O(|C| \cdot r^2 \cdot d^{r+1})$ (see section 2.1.1), and the time complexity of a single iteration of IJGP is $O(g \cdot (n+N) \cdot d^{w^*+1})$, as stated by the theorem 3.4.1. The maximum size of a mini-bucket w^* is always greater or equal than the maximum constraint arity r. This means that the real bottleneck lies in IJGP only. The discussion Gogate and Dechter have performed on the complexity of their sampling algorithm therefore holds for our algorithm as well.

4.4 Interval-IJGP Sampling

As we already noted in the previous section, IJGP performs very slowly on problems with larger domains. When IJGP computes a message from a join-graph node u, it loops over all combinations of values in the domains of variables in the variable label of u, $\chi(u)$. What if we used only some of the combinations to compute the message? Such an approach may decrease the accuracy of the sampling algorithm, but it should improve its run-time performance.

In order to achieve that, we assume that the domains only contain integer numbers (every CSP with finite domains can be mapped to integer-domain CSP). We split the domain of each variable into η intervals (η is a parameter of the algorithm). Then, when computing a message, we loop over combinations of *intervals* rather than values. In order to compute the probability of the combination of intervals (which is indeed necessary for computing the message), we select ζ values from each interval and compute the product of the constraint functions for these values (ζ is another parameter of the algorithm).

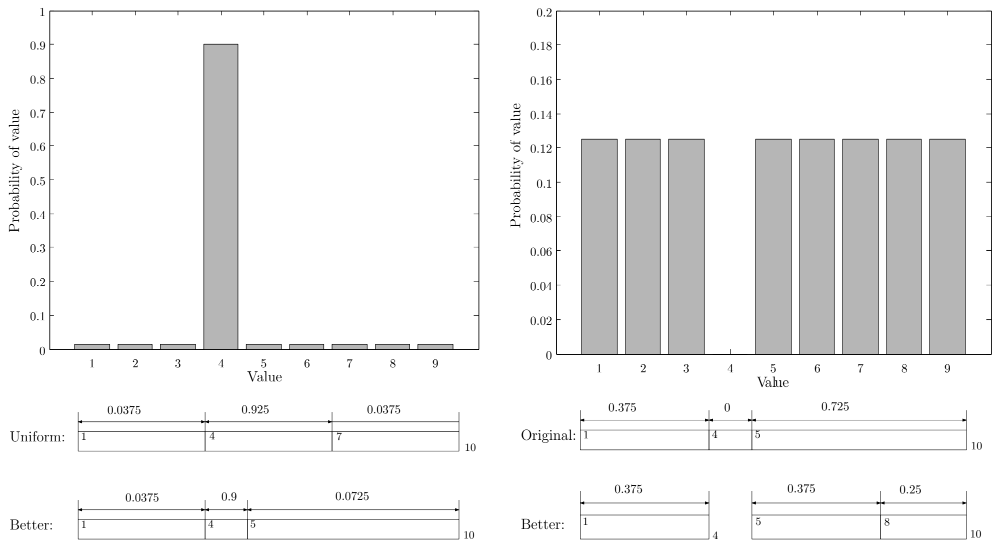
However, splitting domains into intervals of uniform length can lead to poor results. Suppose that we had a CSP problem with two variables: X_1 and X_2. The variable X_1 has the domain $D_1 = \{0, 1\}$ and X_2 has the domain $D_2 = \{1, 2, \ldots, 9\}$. There is one hard constraint $c_1 : (X_1 = 1) \Rightarrow (X_2 \neq 4)$, and one soft constraint c_2, whose constraint function f_{c_2} gives a value 0.9 to any assignment where $X_2 = 4$, and a value 0.00625 to any other assignment. This means that a sampled solution $\overline{a}$ will have probability 0.9 that $\overline{a}[X_2] = 4$ and probability 0.0125 that $\overline{a}[X_2] = x_2$ for any value $x_2 \in D_2, x_2 \neq 4$. The figure 4.1a shows a probability histogram of values of the variable X_2.

Suppose now that we would split the domain into $\eta = 3$ intervals of uniform length, and we would select only $\zeta = 1$ value from the interval during the message computation. The value 4 would have low chance of being used in the message computation, despite its high probability. We would risk loosing the information about the high probability of 4 altogether. Therefore it is better to create intervals $\langle 1, 4)$, $\langle 4, 5)$ and $\langle 5, 10)$ (note that the upper bound is *not* present in the interval). By this we achieve that the high-probability-value 4 is always selected for message computation.

Now suppose that during sampling, we would first set $X_1 := 1$? That would filter the value 4 out from the domain of X_2, because of the first hard constraint c_1. The interval $\langle 4, 5)$ now yields zero probability, and it is essentially useless for the computation. It would be better if we would remove it and split some other interval instead (as suggested by figure 4.1b).

Because of all this, it would be advantageous if the intervals had different length to reflect the estimated probability that values inside an interval will be present in a solution sample. We should also be able to dynamically restructure the intervals as values are filtered out from the variable domains.

To ensure that the length of the intervals really corresponds to the estimated probability of values inside them, we add another piece of information to an interval – an estimated *interval-probability* field.



(a) Initial probabilities and intervals (b) After propagation

Figure 4.1: Dynamic interval length example: Histograms show the probability of each value in the domain of variable X_2. Rectangles below them show intervals and their estimated probabilities. The number in the upper left corner of a rectangle is the lower bound of an interval, the number next to the bottom right corner is its upper bound. The number above the rectangle is the sum of probabilities of values in the corresponding interval.

Ideally, all intervals in an interval-set should have approximately the same estimated interval-probability. However, if there is an interval with one value only, we don't split it even if its estimated probability is much larger than the estimated probabilities of other intervals. By this we make sure that the interval-set captures enough details of the probability distribution over values – for values with higher probability, the intervals around them will be shorter, while for values with small probability, the intervals they are in will be longer.

We define an *interval-set* of a variable X_i as follows:

Definition 4.4.1. *Let X_i be a CSP variable with a linearly ordered integer domain D_i. An* interval-set *for X_i is the set $\mathcal{I} = \{(l_1, u_1, p_1), \ldots, (l_n, u_n, p_n)\}$. The triple (l_j, u_j, p_j) represents an interval $\langle l_j, u_j)$ which has an estimated interval-probability p_j, and $u_j \leq l_k$ for $j < k$, $\forall j : l_j < u_j$ and $p_j \geq 0$.*

An interval-set $\mathcal{I}$ is normalized*, iff*

$$\sum_{(l_j,u_j,p_j)\in\mathcal{I}} p_j = 1 \tag{4.4}$$

We generally remove intervals with estimated interval-probabilities $p = 0$ from the interval-set, though we allow them in the definition, because we need to set the estimated probabilities to 0 prior to their re-computation (described below).

4.4.1 Interval-Set Operators

For interval-sets, we define the following basic operators:

CREATE(c, X_i) Let c be a constraint over scope S_c, $X_i \in S_c$. For every value $x_i \in D_i$, we compute

$$p_{x_i} = \sum_{\substack{\overline{a} \\ \overline{a}[X_i]=x_i}} f_c(\overline{a}) \tag{4.5}$$

where we sum over all assignments $\overline{a}$ of the variables in the scope S_c. Then, we create an interval-set $\mathcal{I}_c = \{(x_i, x_i+1, p_{x_i}) | x_i \in D_i\}$.

The time complexity of **CREATE** is $O(d^{|S_c|})$, where d is the maximum size of a domain for a variable in S_c.

SPLIT($\mathcal{I}$, η) This operator splits every interval $(l, u, p) \in \mathcal{I}$ with $p > 1/2\eta$ uniformly into smaller intervals (l', u', p'), each with estimated probability $p' \leq 1/2\eta$, $p' = \frac{u'-l'}{u-l}$. If the interval length $(u - l)$ is smaller than the required number of splitted intervals, split it into as many intervals as possible (this also means that we don't split intervals of length 1).

Instead of requiring the splitted intervals to have their estimated probability lower than $1/2\eta$, we could use some other constant (such as $1/3\eta$). This allows us to trade-off the complexity and the accuracy of the split.

The goal of the split step is to break intervals which have much larger estimated probability than other intervals. Because of this, other operations (most importantly **JOIN**) can create an interval-set with a more uniformly distributed interval-probability estimation.

See the pseudo-code of **SPLIT** in algorithm 4.3.

Algorithm 4.3 SPLIT($\mathcal{I}$, η)

```
Let $\mathcal{I}' := \emptyset$
Let $p_\eta := 1/\eta$ be the ideal estimated probability an interval could have
for interval $(l, u, p) \in \mathcal{I}$ do
    number of sub-intervals is $n := \max(\min(\lceil 2p/p_\eta \rceil, u - l), 1))$
    $l' := l$
    for $i \in \{1, \ldots, n\}$ do
        $u' := l + \lceil (u - l) \cdot \frac{i}{n} \rceil$
        $\mathcal{I}' := \mathcal{I}' \cup \{(l', u', p \cdot \frac{u'-l'}{u-l})\}$
        $l' := u'$
    end for
end for
Return $\mathcal{I}'$
```

JOIN($\mathcal{I}$, η) For a *normalized* interval $\mathcal{I}$, create a new interval-set $\mathcal{I}'$ with at most η elements.

The **JOIN** algorithm uses heuristic to join neighbouring intervals. See its precise description in an algorithm 4.4, and an example on figure 4.2.

To help spreading the probabilities between intervals as uniformly as possible, we perform a **SPLIT** step before each **JOIN**.

The time complexity of **JOIN** is $O(|\mathcal{I}| + \eta)$.

INTERSECT($\mathcal{I}_1, \mathcal{I}_2$) This operator creates a new interval-set $\mathcal{I}'$ which contains all triples (l, u, p) such that

(i) $\exists i, j : (l_i, u_i, p_i) \in \mathcal{I}_1, (l_j, u_j, p_j) \in \mathcal{I}_2$

(ii) $l = \max(l_i, l_j)$, $u = \min(u_i, u_j)$, $l < u$

(iii) $p = \frac{u-l}{u_i-l_i} \cdot p_i \cdot \frac{u-l}{u_j-l_j} \cdot p_j$, $p > 0$

The time complexity of **INTERSECT** is $O(|\mathcal{I}_1| + |\mathcal{I}_2|)$ – we can intersect intervals by one simultaneous iteration through $\mathcal{I}_1$ and $\mathcal{I}_2$ in an ascending order. Since for every interval set $\mathcal{I}$ defined over variable X_i, it is $|\mathcal{I}| < |D_i|$, an intersection of two interval sets defined over the same variable takes time $O(|D_i|)$. The resulting interval-set $\mathcal{I}$ is not normalized.

Algorithm 4.4 **JOIN**($\mathcal{I}$, η)

```
Let $\mathcal{I}'$ be the resulting interval-set
Let $p_{curr}$ be the probability of currently created interval, $l_{curr}, u_{curr}$ its lower and upper bounds
$p_{curr} := 0$
No interval is currently created
for $(l, u, p) \in \mathcal{I}$ in an order from the lowest interval do
  if no interval is currently created then
    if $p \geq 1/\eta$ then
      $\mathcal{I}' := \mathcal{I}' \cup \{(l, u, p)\}$
    else
      $p_{curr} := p$, $l_{curr} := l$, $u_{curr} := u$
    end if
  else
    if $p_{curr} + p \geq 2/\eta$ then
      $\mathcal{I}' := \mathcal{I}' \cup \{(l_{curr}, u_{curr}, p_{curr}), (l, u, p)\}$
      Unset $l_{curr}, u_{curr}$
      Set $p_{curr} := 0$ (no interval is currently created)
    else
      if $p_{curr} + p \geq 1/\eta$ then
        $\mathcal{I}' := \mathcal{I}' \cup \{(l_{curr}, u, p_{curr} + p)\}$
        Unset $l_{curr}, u_{curr}$
        Set $p_{curr} := 0$ (no interval is currently created)
      else
        $u_{curr} := u$, $p_{curr} := p_{curr} + p$
      end if
    end if
  end if
end for
if some interval was currently created then
  $\mathcal{I}' := \mathcal{I}' \cup \{(l_{curr}, u_{curr}, p_{curr})\}$
end if
```

Intuitively speaking, the intersection should express a joint estimation of interval-probabilities. Figure 4.4 should give the reader an idea what the result of **INTERSECT** looks like.

NORMALIZE($\mathcal{I}$) For an interval-set $\mathcal{I}$, compute the total estimated probability

$$p = \sum_{(l_i, u_i, p_i) \in \mathcal{I}} p_i \tag{4.6}$$

and if $p > 0$, create a new *normalized* interval-set

$$\mathcal{I}' = \{(l_i, u_i, \frac{p_i}{p}) | (l_i, u_i, p_i) \in \mathcal{I}\} \tag{4.7}$$

ADJUST-TO-DOMAIN($\mathcal{I}, D$) Create a new interval-set $\mathcal{I}'$ from $\mathcal{I}$ by adjusting the bounds of intervals in $\mathcal{I}$ so that they fit the domain D. The resulting interval-set $\mathcal{I}'$ contains all intervals (l', u', p') such that

(i) $\exists i : (l_i, u_i, p_i) \in \mathcal{I}$

(ii) $l' = \min_{k \in D}\{k \geq l_i\}$

(iii) $u' = \max_{k \in D}\{k < u_i\} + 1$, $l' < u'$

(iv) $p' = p_i$

This means that for each interval in $\mathcal{I}$ we adjust its lower and upper bound so that there is some value $k \in D$ equal to the lower bound, and some value $k' \in D$ such that $k' + 1$ is equal to the upper bound. The estimated interval-probability remains the same. Intervals which do not contain any values from the domain D are removed from $\mathcal{I}$.

See an example of **ADJUST-TO-DOMAIN** on figure 4.3.

CLEAR($\mathcal{I}$) Set probabilities of intervals in an interval-set $\mathcal{I}$ to 0.

ADD-PROBABILITY($\mathcal{I}, l, u, p$) For an interval $\langle l, u)$, add p to its probability. In other words

$$\mathcal{I} := \begin{cases} (\mathcal{I} \backslash \{(l, u, p')\}) \cup \{(l, u, p' + p)\} & \text{if } (l, u, p') \in \mathcal{I} \\ \mathcal{I} \cup \{(l, u, p)\} & \text{otherwise} \end{cases} \tag{4.8}$$

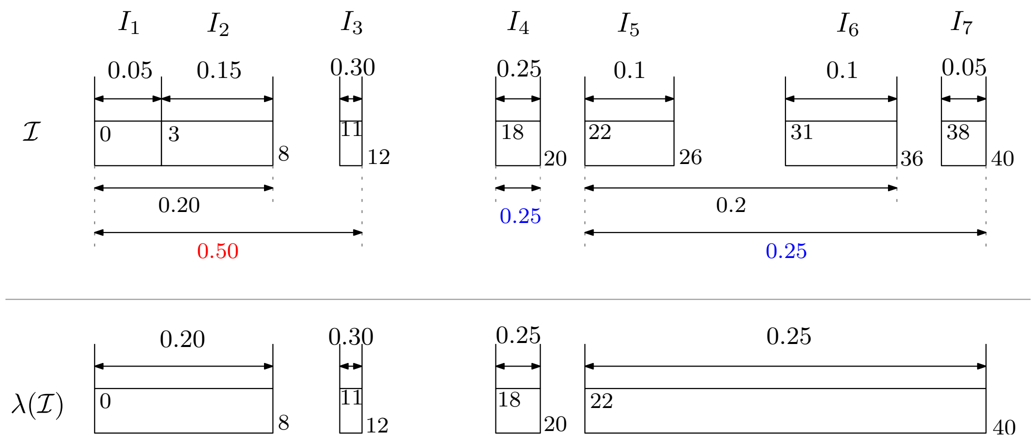


Figure 4.2: Join of an interval-set $\mathcal{I}$ into $\eta = 4$ intervals. Intervals are denoted as rectangles. The number in upper-left corner of the rectangle denotes the lower bound l, the number outside of the bottom-right corner denotes an upper-bound u. An interval rectangle represents an interval $\langle l, u)$. Interval probability is showed above the rectangle. The numbers below rectangles show accumulated probability of the currently joined interval. If the number is red, the accumulated probability has exceeded $2/\eta = 1/2$ and two new intervals were created. If the number was blue, the accumulated probability has exceeded $1/\eta = 1/4$ and one new interval was created.

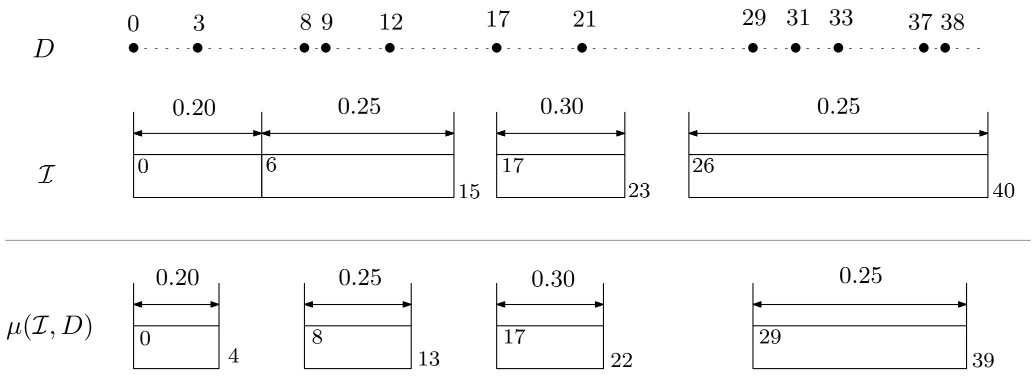


Figure 4.3: Adjust of an interval list $\mathcal{I}$ to a domain D. Elements in D are denoted as large dots. Note that the newly created intervals have the same probabilities as the original ones.

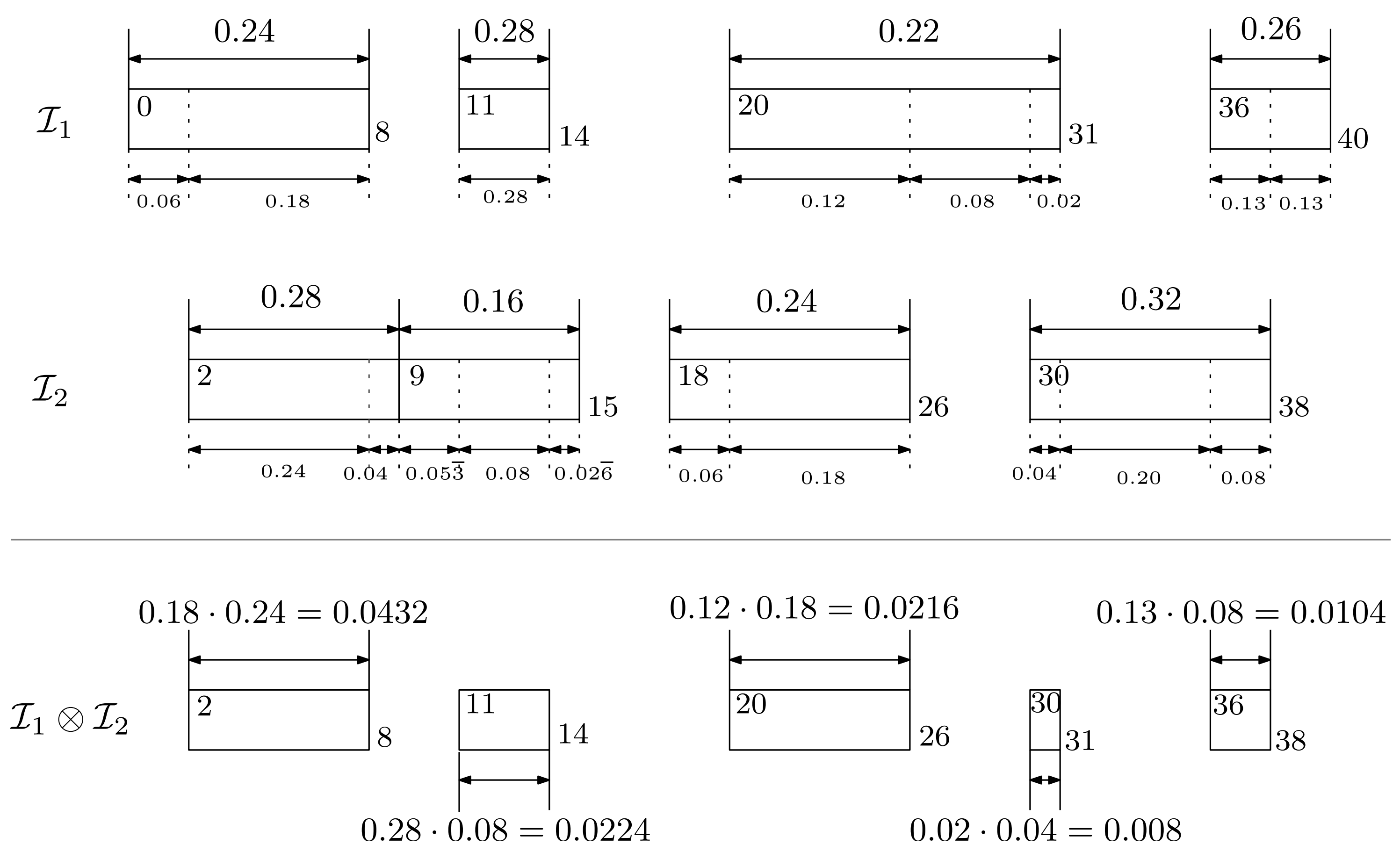


Figure 4.4: Intersection of two interval lists, $\mathcal{I}_1$ and $\mathcal{I}_2$. The dotted lines over original rectangles denote merge points, and the numbers below rectangles denote proportional probability of the corresponding sub-intervals.

4.4.2 Interval-IJGP Algorithm

Prior to running IJGP for the first time, we need to initialize the domain interval-sets. Suppose that we have a join-graph $JG = (G = (V, E), \chi, \psi)$ and a constraint network $\mathcal{R} = (X, D, C)$. For each each constraint $c \in C$, we call **CREATE(c, X_i)** for every variable X_i in the scope S_c. Then for each node $u \in V$ and for every variable X_i in the variable label $\chi(u)$, we perform **INTERSECT** on interval-sets corresponding to that variable from all constraints in the constraint label $\psi(u)$. This creates a new interval-set $\mathcal{I}_{(u,X_i)}$. After that, we normalize $\mathcal{I}_{(u,X_i)}$ and perform **JOIN** to create η intervals.

If the constraint label $\psi(u)$ is empty, we set $\mathcal{I}_{(u,X_i)}$ to be a set of at most η intervals with uniform length: Each interval in such interval-set has equal estimated probability and for any two intervals $I_1, I_2 \in \mathcal{I}_{(u,X_i)}$, their length difference $||I_1| - |I_2|| \leq 1$. If the size of the domain of variable X_i, $|D_i|$, is less than η, we obviously create only $|D_i|$ intervals.

The structure of IJGP algorithm remains generally the same. There are three major changes:

1. Message $m_{(u,v)}$ is no longer a table of probabilities for every combination of values of the variables in $\theta((u, v))$, but rather a table of probabilities for every combination of *intervals* from the interval-sets $\mathcal{I}_{(u,X_1)}, \ldots, \mathcal{I}_{(u,X_n)}$ where $X_1, \ldots, X_n$ are variables from the join-graph edge label $\theta((u, v))$

2. The message update rule changes: It does not loop over values in domains. Instead, it loops over intervals in domain interval-sets.

3. The stopping rule also changes: We perform a fixed number of IJGP iterations, because Kullback-Leibler divergence is not well-defined for interval-sets

Let us discuss these changes in more detail. Remember from the section 3.4 that a message $m_{(u,v)}$ was a function of variables in the join-graph edge label $\theta((u, v))$ of an edge (u, v). We could represent this function as a table of probabilities for every combination of values in the domains of $\theta((u, v))$. Now that we don't use values but intervals, we also have to use a combination of intervals rather than values.

The message update rule becomes more complicated. If we simply tried to replace all references to variable values with variable intervals, we would hit one problem: Different nodes can use different interval-sets for the same variable.

We "workaround" this problem by computing the message as follows: For every variable $X_i \in \chi(u)$, we sum over all intervals in its interval-set $\mathcal{I}_{(u,X_i)}$ as if they were values. Then, for each interval I_i, we randomly select ζ values from $I_i \cap D_i$ and compute the product of constraint functions and message evaluations for these values (ζ is, as noted before, a parameter of the algorithm).

How do we compute a message evaluation for a particular assignment of variables $\overline{a}$ (as messages are now functions of *intervals*)? We introduce a new function $\vartheta(m_{(u,v)}, \overline{a})$ which would evaluate the message $m_{(u,v)}$ for an assignment $\overline{a}$. A simplified version of this message evaluation function for a single-variable-message works as follows:

$$\vartheta(m_{(u,v)}, \overline{a} = (X_i, x_i)) = \tfrac{m_{(u,v)}(I_i)}{|I_i \cap D_i|} \quad x_i \in I_i, I_i \in Dom(m_{(u,v)}) \tag{4.9}$$

where $Dom(m_{(u,v)})$ is the interval-set which constitutes the domain of the message $m_{(u,v)}$. Note that we divide the total interval probability $m_{(u,v)}(I_i)$ with the number of values in that interval $|I_i \cap D_i|$. By this, we compute an approximate probability of a single value.

If we already have some fixed evidence $\mathbf{e}$, we can re-define the previous equation so that it uses the evidence as well:

$$\vartheta(m_{(u,v)}, (X_i, x_i), \mathbf{e}) = \begin{cases} 1 & (X_i, x_i) \in \mathbf{e} \\ 0 & (X_i, x_i') \in \mathbf{e}, x_i \neq x_i' \\ \vartheta(m_{(u,v)}, (X_i, x_i)) & \text{otherwise} \end{cases} \tag{4.10}$$

For a message which is defined over more variables $\mathcal{X}$, we need to find an interval from message domain for every variable in $\mathcal{X}$. The definitions are then analogical.

Recall the message update rule for pairwise Markov random fields (equation 3.19) – we will use it for the sake of simplicity of presentation to illustrate the new message update rule. Using the message evaluation function ϑ, the interval-set-version of this rule looks as follows:

$$\begin{aligned} m_{ji}(I_i|\mathbf{e}) \quad \leftarrow \quad & \textstyle\sum_{x_i \in s(I_i \cap D_i, \zeta)} \sum_{I_j \in \mathcal{I}_j} \sum_{x_j \in s(I_j \cap D_j, \zeta)} \phi(x_j|\mathbf{e}) \cdot \\ & \cdot \psi_{ji}(x_i, x_j|\mathbf{e}) \textstyle\prod_{k \in \mathcal{N}_j \setminus \{i\}} \vartheta(m_{kj}, (X_j, x_j), \mathbf{e}) \end{aligned} \tag{4.11}$$

where the function $s(I_i \cap D_i, \zeta)$ randomly selects ζ values from the intersection of the interval I_i with the domain D_i.

A recursive method to update a join-graph message is presented in algorithm 4.5.

Algorithm 4.5 Interval-IJGP Message Update Rule

```
Let $(u, v)$ be the edge over which we are computing the message

Let $\mathbf{e}$ be an initial assignment of variables (evidence)

Let $\overline{I}$ be the current assignment of intervals to variables; at the beginning,
for each value $x_i$ assigned to a variable $X_i$ by evidence $\mathbf{e}$, if $X_i \in \theta((u, v))$,
assign to $X_i$ an interval $I$ from interval-set $\mathcal{I}_{(u,X_i)}$ such that $I$ contains $x_i$
At the beginning (prior to recursive calls), set $m_{(u,v)}(\overline{I}') := 0$ for all pos-
sible interval assignments $\overline{I}'$

Let $\overline{a}$ be the current assignment of values to variables; start with an $\overline{a} = \mathbf{e}$

if all variables in the variable label $\chi(u)$ have been assigned in $\overline{a}$ then
  $p := \prod_{f_c \in \psi(u)} f_c(\overline{a}[S_c]) \prod_{\substack{m_{(w,u)} \\ w \neq v}} \vartheta(m_{(w,u)}, \overline{a}, \mathbf{e})$
  {update the message probability for interval assignment $\overline{I}$}
  $m_{(u,v)}(\overline{I}) := m_{(u,v)}(\overline{I}) + p$
  {Re-compute the estimated interval probabilities}
  for each $X_i \in \chi(u)$ do
    Let $\langle l, u)$ be an interval in $\mathcal{I}_{(u,X_i)}$ which contains the value $x_i$ assigned
    to $X_i$ by the assignment $\overline{a}$
    ADD-PROBABILITY($\mathcal{I}_{(u,X_i)}, l, u, p$)
  end for
else
  Let $X_j$ be the currently processed variable from $\chi(u)$; $X_j$ was not yet
  assigned any value in the assignment $\overline{a}$
  for interval $I_k \in \mathcal{I}_{(u,X_j)}$ do
    if $X_j \in \theta((u, v))$ then
      Assign interval $I_k$ to $X_j$ in $\overline{I}$
    end if
    $V := s(I_k \cap D_j, \zeta)$ {Select $\zeta$ random values from $I_k \cap D_j$)}
    for $x_j \in V$ do
      Assign $x_j$ to $X_j$ in assignment $\overline{a}$
      Recursively call message update rule to process next variable
      Remove $X_j = x_j$ from $\overline{a}$
    end for
  end for
end if
```

Recomputing Domain Interval-Sets

The algorithm described above did not take into account the interval probability changes brought in by messages incoming into a join graph node. In an attempt to tackle this issue, we added one step: we recompute the probabilities of intervals in domain interval-sets as we compute new message.

Suppose that we are computing a message $m_{(u,v)}$ for a join-graph node u. Prior to computing it, we call **CLEAR**$(\mathcal{I}_{(u,X_i)})$ on every interval-set $\mathcal{I}_{(u,X_i)}$ for every variable $X_i \in \chi(u)$. We already observed that in order to compute the message, we need to loop over all combinations of intervals from interval sets of variables in $\chi(u)$. Now suppose that we have just computed the product p of constraint functions and message evaluations for a variable assignment with values from these intervals. We call **ADD-PROBABILITY**$(\mathcal{I}_{(u,X_i)}, l_i, u_i, p)$ for every interval (l_i, u_i, p') of variable X_i. By doing this, we sum all probabilities relevant for a given interval.

After we compute the new message, we normalize all interval-sets $\mathcal{I}_{(u,X_i)}$ for all $X_i \in \chi(u)$.

4.4.3 Time Complexity of Interval-IJGP

Theorem 4.4.1. *The initialization of interval-sets for the Interval-IJGP algorithm has the time complexity*

$$O(|C| \cdot r \cdot d^r + N \cdot w^* \cdot (d + \eta)) \tag{4.12}$$

where $|C|$ is the total number of constraints, r the maximum constraint arity, d the maximum domain size, N the number of nodes in the join-graph, w^* the maximum size of a variable label of a join-graph node, and η is the algorithm parameter (required number of intervals per domain).

Proof. During the initialization of the interval-sets, we call **CREATE** for every constraint and every variable in its scope. This has the time complexity $O(|C| \cdot r \cdot d^r)$.

After that, we need to intersect interval-sets created for every constraint within every join-graph node. This means that for each constraint, we intersect an interval-set created in the previous step for every variable in its scope. Therefore, we need to perform $O(|C| \cdot r)$ intersects, each with time complexity $O(d)$.

For every join-graph node u and for every variable $X_i \in \chi(u)$, we need to join the intersected intervals created in the node u for variable X_i. This has the time complexity $O(N \cdot w^* \cdot (d+\eta))$, since the worst time complexity of **JOIN** is $O(d+\eta)$. $\square$

Theorem 4.4.2. *The time complexity of the* Interval-IJGP message update rule *is*

$$O((\eta \cdot \zeta)^{w^*} \cdot (w^* \cdot \zeta \cdot \log d) \cdot (\Psi + g \cdot w^* \cdot \log \eta + w^*)) \tag{4.13}$$

where Ψ is the maximum size of a constraint label $\psi(u)$ for any join-graph node u and g is the maximum degree of a join-graph node.

Proof. Recall from the message update algorithm 4.5 that we loop over assignments of ζ values from η intervals for each variable in the variable label $\chi(u)$ of a join-graph node u. There are $(\eta \cdot \zeta)^{w^*}$ such assignments. This is the first item of the product in the theorem.

For each interval I in the interval assignment, random selection of ζ values from $I \cap D_i$ can be done in time $O(\zeta \cdot \log d)$ (the domain D_i can be stored as a tree, and we can randomly select branches in which we are going to look for values; we perform ζ such selections). As there are at most w^* intervals in the assignment (one for each variable in $\chi(u)$), random selection of the values from intervals is going to take us the time $O(w^* \cdot \zeta \cdot \log d)$. This constitutes the second item of the product in the theorem.

For each assignment of values $\overline{a}$, we need to compute the product of constraint functions and message evaluations. There are at most Ψ constraints in the constraint label, and at most g incoming messages.

For each incoming message m, we need to compute its evaluation for the assignment of values $\overline{a}$. Suppose that a variable X_i in the scope of the message has been assigned the value x_i. We need to find an interval in the interval-set the message m uses for variable X_i, such that it contains the value x_i. This can be done in time $O(\log \eta)$. We need to do this for every variable in the scope of message m, and there are at most w^* such variables. Together, computing message evaluations is going to take us the time $O(g \cdot w^* \cdot \log \eta)$.

Also, for each assignment of values and intervals, we recompute the estimated interval probabilities by calling **ADD-PROBABILITY** on a corresponding interval in an interval-set $\mathcal{I}_{(u,X_i)}$ for every variable $X_i \in \chi(u)$. This completes the third item of the product in the theorem: $O(\Psi + g \cdot w^* \cdot \log \eta + w^*)$. $\square$

In one iteration of the Interval-IJGP algorithm, we send a message over each join-graph edge in both directions. Therefore the following theorem is a corollary of the previous theorem:

Theorem 4.4.3. *One iteration of the Interval-IJGP algorithm has the time complexity*

$$O(|E| \cdot (\eta \cdot \zeta)^{w^*} \cdot (w^* \cdot \zeta \cdot \log d) \cdot (\Psi + g \cdot w^* \cdot \log \eta + w^*)) \tag{4.14}$$

where $|E|$ is the number of edges in the join-graph.

Compare the previous theorem with the theorem 3.4.1 (time complexity of a single iteration of IJGP): For IJGP, the time complexity mostly depends on d^{w^*}. For Interval-IJGP, it depends mostly on $(\eta \cdot \zeta)^{w^*}$. This means that using Interval-IJGP only makes sense if $\eta \cdot \zeta$ is lower than d, enough to compensate for other overhead introduced by the interval representation.

4.4.4 Interval-IJGP + GAC Sampling

Similarly to plain IJGP sampling, we can use generalized arc-consistency with Interval-IJGP algorithm, too.

The main scheme of the sampling algorithm looks analogous to algorithm 4.2 (the original IJGP + GAC sampling algorithm). There are two essential differences:

1. When we propagate the constraints and remove values from the domains, we call **ADJUST-TO-DOMAIN**$(\mathcal{I}, D)$ on every interval-set $\mathcal{I}$ corresponding to a variable with domain D. After that, we perform a **JOIN** on this interval-set.

2. The probability distribution returned by Interval-IJGP is defined over intervals of values rather than on values itself. Therefore we first pick a random interval I according to this distribution. Then, we randomly select a value from the intersection of I with the domain D of the corresponding variable. When there are no more value from that intervals to try, we remove the entire interval from the distribution.

The development of the Interval-IJGP algorithm arose from observations of the IJGP behaviour on existing benchmark problems. Similarly to the case of the original IJGP algorithm, there is still very little theoretical understanding about why and on what classes of problems the algorithm works. We demonstrate its practical performance on several problems in the following chapter.

Chapter 5

Experimental Results

We have tested the performance of all three algorithms described in the previous chapter on problems from *CostFunctionLib*, a soft constraint benchmark library of INRA Toulouse, France ([Sch]) and on a special instance of *all-interval-series* problem, suggested by Intel for use in their test generator for hardware designs.

All algorithms were implemented in C++, and all tests were run on an Intel Pentium M 1.73 GHz laptop with 1024 MB RAM, running GNU/Linux 2.6.24.

5.1 Problems Tested

5.1.1 CostFunctionLib Problems

The CostFunctionLib contains several classes of *Weighted-CSP* problems (see section 2.2 for the definition). We have used the following classes:

Queen-graph coloring ([Chv]) Given a $n \times n$ chessboard, a queen-graph is a graph on n^2 nodes, each corresponding to a square of the board. Two nodes are connected by an edge if the corresponding squares are in the same row, column, or diagonal. The decision version of the coloring problem asks if it is possible to color the nodes of the graph with k colors so that no two neighbour nodes share the same color (a graph with that property is said to a have a *coloring number* k). A relaxed version of the problem asks if it is possible to find a coloring by k colors such that the number of violations (i.e. edges whose nodes share the same color) is minimized.

The coloring problem has a straightforward explanation for queen-graphs: For an $n \times n$ chessboard, it is possible to place n sets of n queens on the board so that no two queens of the same set are in the same row, column or diagonal, iff the corresponding queen-graph has coloring number n.

Warehouse assignment ([BW58]) In the warehouse assignment problem, a company considers opening warehouses at some locations in order to supply its existing stores. Each store must be supplied by exactly one open warehouse. There is a supply cost describing how expensive it is for a particular warehouse to supply a store. The goal is to find an assignment of n stores to m warehouses such that the total supply costs are minimized.

Weighted CNF formulas ([Hoo00]) In weighted CNF (*conjunctive normal form*) satisfiability problem, we assign each clause in a boolean formula in CNF some cost (a clause is a disjunction of propositional variables). The goal is to find an assignment of the propositional variables such that the total cost of *unsatisfied* clauses is minimized.

In the CostFunctionLib, these problems are described as weighted-CSPs. For each constraint c, we assign a cost to each tuple in the scope of the constraint. There is also a given upper bound on the required cost of a solution. Tuples whose cost is larger than this upper bound are not allowed at all (this means that they are in fact hard constraints). The CostFunctionLib uses the *WCSP* file format to describe its instances (see [Ott] for its description).

We describe the probability distribution for weighted-CSP solutions using the notion introduced in 4.1. Suppose that for a constraint c, we have a corresponding cost function $g_c : \times_{X_i \in S_c} Dom(X_i) \rightarrow \mathbb{R}^+ \cup \{0\}$, where $Dom(X_i)$ is the domain of variable X_i (see the definition of weighted-CSP in section 2.2). We define the probability distribution using a constraint function f_c introduced in equation 4.2. Note that because we try to minimize the cost, we need to select the base of an exponent $0 < \sigma < 1$.

5.1.2 All-Interval-Series

The original idea behind the All-Interval-Series problem comes from music theory. A *series* is a sequence of twelve tone names where each tone occurs exactly once. In our setting we represent the tones as numbers $X =$

$\{0, 1, \ldots, 11\}$ and we look for a permutation of X. In addition to that, all intervals (differences between neighbouring numbers) should be different. In the relaxed version of the problem, the last requirement is a soft constraint. The goal is to find such series which maximizes the number of different interval lengths. This problem can be mapped to various scenarios in test generation for HW designs.

We have used two different SoftCSP models of the problem: A naïve approach, and an approach with extra variables.

In the first model, there are only 12 variables (one for each number, or "tone" in the musical terminology), $V_1 = \{X_1, X_2, \ldots, X_{12}\}$, all of which have domain $D_1 = \{0, 1, \ldots, 11\}$. The constraints are as follows:

$$\forall i \in \{1, \ldots, 11\} \; \forall j \in \{i+1, \ldots 11\} \; |X_i - X_{i+1}| \neq |X_j - X_{j+1}| \tag{5.1}$$

Note that we don't use any symmetry constraints; each inequality is expressed only once.

In the second model, there are 11 additional interval-length-variables $V_2 = \{X_{13}, \ldots, X_{23}\}$, one for each interval between neighbouring numbers. These new variables all have the domain $D_2 = \{1, 2, \ldots, 11\}$. There are two types of constraints: first, difference between neighbouring numbers is hard-constrained to be the same as the value of one additional variable from V_2. Formally, they can be described as follows:

$$\forall i \in \{1, \ldots, 11\} : \; |X_i - X_{i+1}| = X_{i+12} \tag{5.2}$$

Then, there are soft-constraints which denote inequalities between the interval-length-variables:

$$\forall i \in \{13, \ldots, 23\} \; \forall j \in \{i+1, \ldots, 23\} : \; X_i \neq X_j \tag{5.3}$$

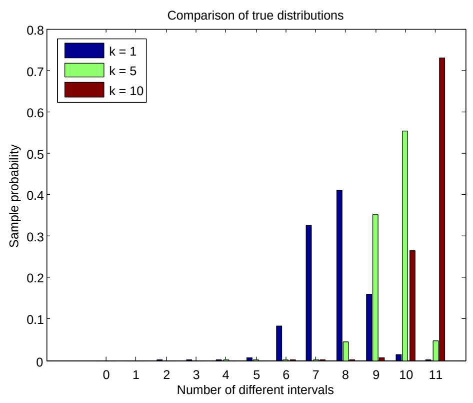
We also add hard-constraints that require the solution to be a permutation of $\{0, 1, \ldots, 11\}$:

$$\forall i \in \{1, \ldots, 12\} \; \forall j \in \{i+1, \ldots, 12\} : \; X_i \neq X_j \tag{5.4}$$

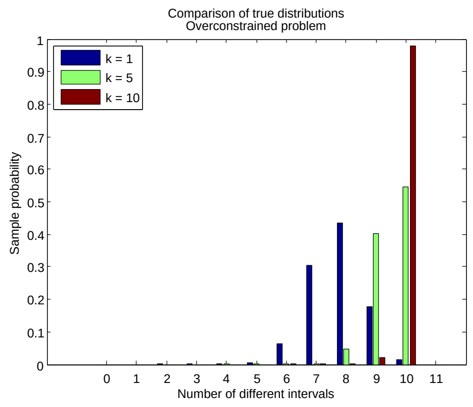
For both models, we have also tested an over-constrained version of the problem. Two constraints were added: $X_1 = 0$ and $X_2 = 10$, so that no solution with 11 different intervals exists.

For the all-interval-series problem, we used the following definition of soft constraint function (required for specifying the target probability distribution):

$$f_i(\mathbf{S}_i) = \begin{cases} 2^k & \text{constraint } c_i \text{ is satisfied} \\ 1 & \text{otherwise} \end{cases} \tag{5.5}$$



(a) Original problem



(b) Over-constrained problem

Figure 5.1: All-interval-series problem: True distribution histograms of the number of different intervals for three different values of the parameter k

where k is a parameter.

The true distributions over the space of all permutations of $\{0, \dots, 11\}$ for different values of k are displayed on figure 5.1. The histograms show the probability of obtaining a solution with given number of different intervals. It can be seen that with increasing k, the chance of sampling mostly from optimal solutions increases.

5.2 CostFunctionLib Tests

We have used the following instances of the problems in CostFunctionLib:

Warehouse assignment instances `warehouse0` and `warehouse1`. The first instance assigns 10 stores to 5 warehouses and the second one assigns 20 stores to 19 warehouses.

Queen-graph-coloring instances `queen5_5_5` and `queen6_6_12`. The first is a task of coloring a 5×5 queen-graph with 5 colors, and the second is a coloring of 6×6 queen-graph with 12 colors.

Two random CNF instances A small non-satisfiable instance `cnf_small`, and a large non-satisfiable instance `cnf_large`.

The CSP-related properties of these instances are specified in table 5.1.

Instance	# variables	max. domain size	# constraints
`warehouse0`	15	5	65
`warehouse1`	39	19	419
`queen5_5_5`	25	5	160
`queen6_6_6`	36	12	290
`cnf_small`	50	2	98
`cnf_large`	300	2	800

Table 5.1: Instances of problems from CostFunctionLib

We compare the performance of our three algorithms with an approximation of the "correct" probability distribution. For weighted-CSP problems, we obtain this approximation as follows: We generate a fixed number of completely random samples from the domains of the CSP. We compute the relative likelihood p_s of each sample s using an equation similar to equation 4.1:

$$p_s = \prod_{c \in C} f_c(s[S_c]) \tag{5.6}$$

where C is the set of all constraints, S_c is the scope of constraint c and f_c is the constraint function.

For each variable $X_i \in X$, we compute its approximate probability distribution as

$$\mathcal{P}(X_i = x_i) = \frac{\sum_{s \in \mathcal{S}_{x_i}} p_s}{\sum_{s \in \mathcal{S}} p_s}$$

where $\mathcal{S}_{x_i}$ is the set of all generated samples in which $X_i = x_i$, and $\mathcal{S}$ is the set of all generated samples.

For each instance in the test we generated 10^6 random samples to approximate the true distribution. Note however that for problems which admit a few feasible solutions, this gives an extremely inaccurate approximation.

5.2.1 Performance Criteria

To make our results comparable with the previous work on CSP sampling using IJGP ([GD06]), we use similar performance criteria to assess the performance of our algorithms.

Given a constraint network $\mathcal{R} = (X, D, C)$, we compute an approximate partial probability distribution for every variable $X_i \in X$: $\mathcal{P}_a(X_i = x_i) =$

$\frac{N_{x_i}}{N}$, where N_{x_i} is the number of samples which assign the value x_i to X_i, and N is the total number of samples.

We use the following criteria to compare the distributions:

Kullback-Leibler divergence See its definition in section 2.3. For each variable $X_i \in X$, we compute the KL-divergence between the approximate partial probability distribution $\mathcal{P}_a$ generated by our algorithm, and the "true" distribution approximation. Then we compute an average KL-divergence over all variables in X. The smaller the KL-divergence, the closer the resulting probability distribution matches the target distribution.

Variance of the samples We compute sample variance for every variable

$$S_{m_i}^2 = \frac{1}{m-1}\sum_{j=1}^{m}(X_{j_i} - \overline{X}_i)^2$$

where m is the number of generated samples, X_{j_i} is the value of variable X_i in the j-th sample, and $\overline{X}_i$ is the average value of variable X_i over all generated samples. Then we compute an average of the variance over all variables.

As we have already mentioned in section 3.5, test generation does not really require *uniform* samples; in practice, it is useful just to have samples different enough from each other. Therefore the larger the variance, the better (from the test generation perspective).

Average cost We compute an average total cost of all generated samples. Note that by *cost* we mean cost in weighted CSP sense: Sum of costs each constraint assigned to the sample. This means that the cost of a sample does *not* depend on any of the parameters of our SoftCSP probability distribution. For weighted CSP, the lower the cost, the better the solution.

5.2.2 Results and Observations

Recall that from the definition of SoftCSP probability distribution (section 4.1), the proportion of probability given to low-cost solutions depends on the parameter σ, and for weighted-CSP, we should choose $0 < \sigma < 1$. We have performed all our tests on CostFunctionLib instances with two different

	Gibbs	IJGP	Interval-IJGP			
			$\eta = 1$ $\zeta = 1$	$\eta = 1$ $\zeta = 2$	$\eta = 3$ $\zeta = 1$	$\eta = 3$ $\zeta = 2$
	#S time (s) KL Var Cost	#S time (s) KL Var Cost	#S time (s) KL Var Cost	#S time (s) KL Var Cost	#S time (s) KL Var Cost	#S time (s) KL Var Cost
$\sigma = \frac{1}{2}$	1000 114 34.1977 0.832 9.9	50 6 29.2216 0.517 8.3	50 14 36.0873 1.399 21.5	50 21 36.0962 1.485 23.4	50 29 28.9443 0.156 4.4	50 114 28.3633 0.138 4.3
$\sigma = \frac{1}{2}^{0.01}$	1000 113 0.1878 1.401 22.3	50 5 0.2103 1.422 22.2	50 14 0.2225 1.399 21.5	50 21 0.1879 1.485 23.4	50 29 1.0792 3.721 24.8	50 113 1.1429 3.377 23.8

Table 5.2: Results on `warehouse0` instance. Each cell contains the number of samples generated (#S), the run-time in seconds to generate all samples, Kullback-Leibler divergence between the distribution of samples and the "correct" distribution (KL), average variance of samples (Var) and average cost of samples (Cost).

values of σ: $1/2$ and $(1/2)^{0.01}$. The first choice of σ gives higher probability to low-cost solutions than the second choice. The probability distribution for $\sigma = (1/2)^{0.01}$ is "smoother".

We have run Gibbs sampling and IJGP sampling on all tested instances. For Gibbs sampling, we have set the length of burn-in to 10000 for all instances. In IJGP sampling we have set the maximum bucket size to 3 for all instances (recall from section 3.4 that the maximum bucket size determines the number of variables within a single node of a join-graph). Interval-IJGP sampling has been run on all instances except of the two CNFs, since the algorithm is primarily targeted towards domains larger than binary. We have run Interval-IJGP with several combinations of parameters η, ζ to observe their influence on algorithm accuracy and run-time.

Results for the `warehouse0` instance are summarized in table 5.2 and for `warehouse1` they are summarized in table 5.3. For the `warehouse1` instance, we were not able to generate enough feasible solutions by random guessing. Therefore, we did not have an approximation of correct distribution and we had to omit the Kullback-Leibler divergence from the results. On `warehouse0`, all algorithms failed to capture the target distribution for $\sigma = 1/2$ – note the big value of KL-divergence in their results. Nevertheless, they produced samples with lower average cost for $\sigma = 1/2$ that they did for $\sigma = (1/2)^{0.01}$. Gibbs sampling was the fastest and arguably the most accurate algorithm.

The queen-graph results are summarized in tables 5.4 for `queen6_6_12` and 5.5 for `queen5_5_5`. Again, Gibbs sampling wins performance-wise, though for `queen5_5_5` the Interval-IJGP algorithm with $\eta = 1$ and $\zeta = 1, 2$ had better accuracy than Gibbs sampling on the same instance.

The CNF instances proved to be the most suitable for both Gibbs and IJGP sampler. Gibbs sampling had much worse run-time on the large instance, as it was more affected by the large number of constraints. For complete results on CNF instances, see table 5.6.

Gibbs sampling was by far the best performing algorithm from the three ones we have tested. It was able to generate samples much faster than IJGP-based algorithms. It managed to capture the change in distribution with different values of σ – note the decreasing average cost with decreasing values of σ. The IJGP-based algorithms mostly kept the average cost the same, with the only exception of Interval-IJGP sampler on warehouse instances. Gibbs sampling had usually higher variance (though the differences are less significant), making it more useful for test generation tasks (where we need samples to be quite different). With the sole exception of `warehouse0` (where all algorithms performed very badly for $\sigma = 1/2$), Gibbs sampling was usually reasonably close to the target distribution.

Interval-IJGP had usually outperformed pure IJGP in terms of run-time for instances with larger domains. This is to be expected, as the slow run-time performance of IJGP was the primary cause for introducing the interval representation. However, Interval-IJGP had usually lower (or at least comparable) KL-divergence to IJGP, which means that it actually fitted the target distribution better. IJGP sampled solutions with lower average cost, but its variance in solutions was lower than for Interval-IJGP.

As we increased the number of domain intervals η, the variance for Interval-IJGP increased while the average cost decreased (the only exception was the `queen5_5_5` instance). However, we haven't been able to generate enough samples to make this difference significant.

	Gibbs	IJGP	Interval-IJGP					
			$\eta = 4$ $\zeta = 1$	$\eta = 4$ $\zeta = 2$	$\eta = 8$ $\zeta = 1$	$\eta = 8$ $\zeta = 2$	$\eta = 12$ $\zeta = 1$	$\eta = 12$ $\zeta = 2$
	#S time (s) Var Cost	#S time (s) Var Cost	#S time (s) Var Cost	#S time (s) Var Cost	#S time (s) Var Cost	#S time (s) Var Cost	#S time (s) Var Cost	#S time (s) Var Cost
$\sigma = \frac{1}{2}$	1000 5374 1.123 26.2	10 3471 12.507 112.3	10 512 10.024 75.7	10 2499 11.238 71.6	10 1423 6.923 63.0	10 7830 7.219 59.8	10 2128 10.069 81.7	10 11471 7.848 74.6
$\sigma = \frac{1}{2}^{0.01}$	1000 5329 15.567 184.5	10 3312 12.291 113.6	10 479 8.071 161.0	10 2707 7.885 165.2	10 1024 14.472 139.0	10 7148 13.253 145.7	10 2174 11.362 139.5	10 12575 10.464 132.8

Table 5.3: Results on `warehouse1`

		Gibbs	IJGP	Interval-IJGP					
				$\eta = 2$ $\zeta = 1$	$\eta = 2$ $\zeta = 2$	$\eta = 4$ $\zeta = 1$	$\eta = 4$ $\zeta = 2$	$\eta = 6$ $\zeta = 1$	$\eta = 6$ $\zeta = 2$
$\sigma = \frac{1}{2}$	#S	1000	10	10	10	10	10	10	10
	time (s)	4615	10924	189	647	686	4705	1819	14499
	KL	0.1779	2.0967	0.9870	1.0813	1.1681	1.1218	1.1963	1.1598
	Var	13.936	5.903	10.689	10.701	10.307	10.072	11.475	10.504
	Cost	4.8	37.3	121.6	115.4	117.4	115.4	97.6	94.2
$\sigma = \frac{1}{2}^{0.01}$	#S	1000	10	10	10	10	10	10	10
	time (s)	5160	11372	190	649	690	4651	1802	13689
	KL	0.0085	2.1010	0.9831	1.0866	1.1758	1.1224	1.2084	1.1422
	Var	11.981	5.903	10.822	10.649	10.272	10.077	11.710	10.759
	Cost	122.3	37.3	120.5	116.6	118.4	115.4	99.4	99.0

Table 5.4: Results on `queen6_6_12` graph coloring problem

	Gibbs	IJGP	Interval-IJGP			
			$\eta = 1$ $\zeta = 1$	$\eta = 1$ $\zeta = 2$	$\eta = 3$ $\zeta = 1$	$\eta = 3$ $\zeta = 2$
	#S time (s) KL Var Cost	#S time (s) KL Var Cost	#S time (s) KL Var Cost	#S time (s) KL Var Cost	#S time (s) KL Var Cost	#S time (s) KL Var Cost
$\sigma = \frac{1}{2}$	1000 696 0.3580 1.956 6.2	50 1199 0.6797 1.446 34.4	50 145 0.0623 1.961 62.7	50 233 0.0576 1.973 61.5	50 490 0.6633 1.232 94.5	50 2534 0.6701 1.152 97.8
$\sigma = \frac{1}{2}^{0.01}$	1000 705 0.0037 2.008 62.3	50 1198 0.5983 1.507 38.3	50 148 0.0593 1.961 62.7	50 232 0.0581 1.973 61.5	50 504 0.6601 1.211 102.5	50 2571 0.7223 1.138 109.4

Table 5.5: Results on `queen5_5_5` graph coloring problem

	Gibbs	IJGP
	#S time (s) KL Var Cost	#S time (s) KL Var Cost
$\sigma = \frac{1}{2}$	1000 437 0.0310 0.2461 3.286	50 551 0.5623 0.1103 1.360
$\sigma = \frac{1}{2}^{0.01}$	1000 438 0.0213 0.2501 6.198	50 550 0.3310 0.1667 4.580

(a) `cnf_small`

	Gibbs	IJGP
	#S time (s) KL Var Cost	#S time (s) KL Var Cost
$\sigma = \frac{1}{2}$	1000 23063 0.0597 0.2437 24.217	50 1556 0.0512 0.2497 49.380
$\sigma = \frac{1}{2}^{0.01}$	1000 22448 7.51e-4 0.2499 49.703	50 1557 0.0155 0.2496 49.380

(b) `cnf_large`

Table 5.6: Results on CNF instances

5.3 All-Interval-Series Tests

5.3.1 Performance Criteria

We have used two performance criteria: The number of different interval lengths in a series and the time necessary to generate samples. We compared the results with a true distribution over all permutations of numbers $\{0, 1, \ldots, 11\}$ (figure 5.1).

5.3.2 Naïve Encoding: Results

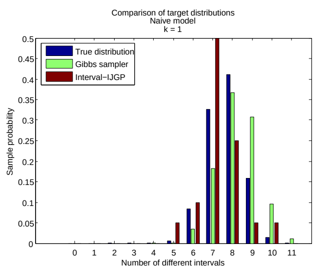
The IJGP-based algorithms (IJGP+GAC and Interval-IJGP+GAC) performed very slowly on the naïvely encoded problem. We even had to remove the IJGP+GAC algorithm from the test, as it was too slow to generate any significant number of solutions. We set the maximum bucket size to 3. However, because the maximum arity of a single constraint was 4, the real bucket size was also 4. This hindered the performance significantly, because the performance of the IJGP algorithm is exponential in the maximum bucket size (as we have seen in section 3.4). We therefore generated only 20 samples for the Interval-IJGP algorithm.

For Gibbs sampling, we generated 1000 samples. Here the time required for sampling was determined mostly by the length of the burn-in, i.e. how many samples were initially thrown away before we declared the samples as "correct". The burn-in length was 10000.

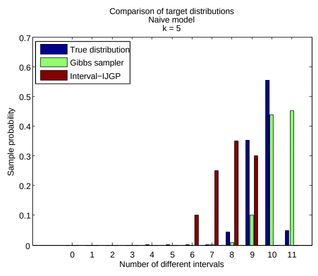
Results are summarized on figure 5.2 for the original problem, and on figure 5.3 for the over-constrained problem.

Gibbs sampling generally produced samples with a high number of different intervals, and for $k = 10$, it even produced samples with an optimal number of intervals. However, it failed to produce a solution that would indeed be a permutation. We tried to add hard-constraints $X_i \neq X_j$ for $i \neq j$, but Gibbs sampling got stuck on a single randomly generated initial solution, and it did not change the solution at all. The reason for this was that for a solution which is a permutation, no neighbouring solution is a permutation (just changing one value breaks the permutation requirement). Gibbs sampling would have to use a different neighbourhood, such as samples created by swapping the values of two variables.

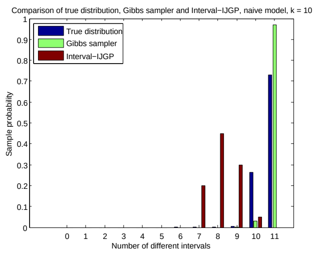
As Gibbs sampling neglected the permutation requirement, it is not surprising that on the over-constrained problem (where no permutation with 11



(a) k = 1

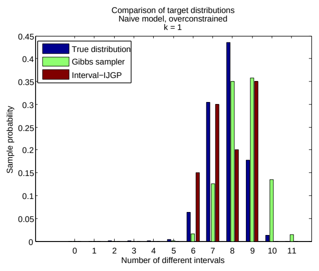


(b) k = 5

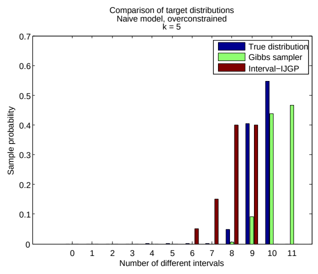


(c) k = 10

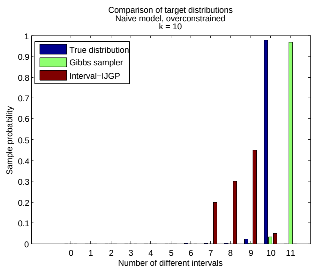
Figure 5.2: All-interval-series, number of different interval lengths for naïve encoding



(a) k = 1



(b) k = 5



(c) k = 10

Figure 5.3: All-interval-series, number of different interval lengths for naïve encoding, over-constrained problem

different intervals exists such that $X_1 = 0$ and $X_2 = 10$) it actually produced samples that had 11 different intervals.

The Interval-IJGP algorithm did not find any optimal solution for any of the parameter choices. As we already observed on CostFunctionLib benchmarks, when the target distribution is “flat”, Interval-IJGP captures it quite well. However, when the target distribution gives higher probability to better solutions (with increasing values of k), the solutions sampled by Interval-IJGP are only slightly better. Notice the fact that the worst solutions (with the number of different intervals equal to 5) are not present in the samples for $k = 5, 10$. It appears that Interval-IJGP has difficulties in targeting the optimal solutions if there are only a few of them.

Algorithm	Time (min)
Gibbs (1000 samples)	1:56
Interval-IJGP + GAC (20 samples)	40:19

Table 5.7: All-interval-series, naïve encoding – Run-time

Algorithm	Time (min)
Gibbs (1000 samples)	3:35
IJGP + GAC (25 samples)	69:01
Interval-IJGP + GAC (50 samples)	39:54

Table 5.8: All-interval-series, encoding with additional variables – Run-time

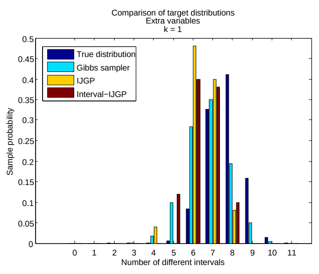
5.3.3 Encoding with Additional Variables: Results

For the encoding with additional variables, the Interval-IJGP algorithm was faster than for the naïve encoding, but its accuracy remained basically the same. The results are summarized on figure 5.4 for the original problem, and on figure 5.5 for the over-constrained problem.

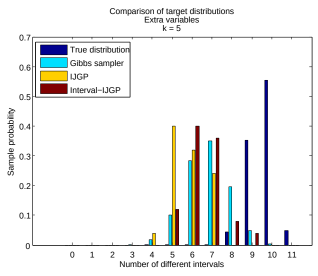
The Interval-IJGP algorithm had slightly larger variance than the pure IJGP one, and in general it produced samples with higher number of different interval lengths. However, as in the case of CostFunctionLib instances, both algorithms failed to adjust their samples to changes in the target distribution: With increasing k, the average number of different intervals remained almost the same.

Gibbs sampler had significant troubles handling the hard constraints. Once it found a solution, it was hard for it to move to another solution, because all "neighbour" solutions had zero probability. To help the Gibbs sampler, we adjusted the constraint function for hard constraints so that in case the constraint is not satisfied, it still has some small value (e.g. 10^{-20}). Still, for $k = 10$ it got stuck in a single solution which had reasonable fitness (therefore the spike in distribution on figure 5.4).

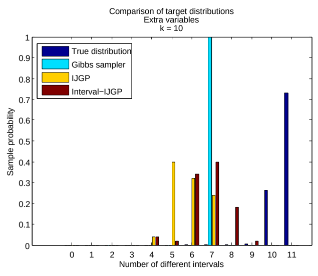
The results for the over-constrained problem were more-or-less similar to the original problem. The only exception was the Gibbs sampling, which managed to avoid getting stuck in local maximum for $k = 10$.



(a) k = 1

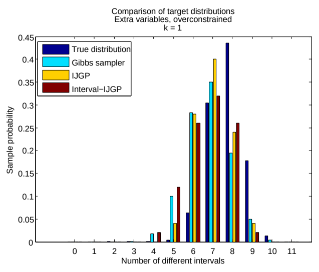


(b) k = 5

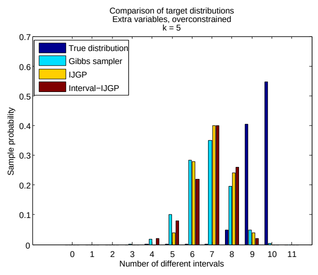


(c) k = 10

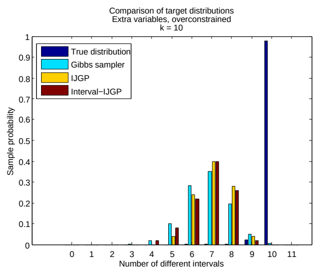
Figure 5.4: All-interval-series, number of different interval lengths for encoding with extra variables



(a) k = 1



(b) k = 5



(c) k = 10

Figure 5.5: All-interval-series, number of different interval lengths for encoding with extra variables, over-constrained problem

Chapter 6

Conclusions

We have focused on a specific problem which occurs when constraint satisfaction techniques are used to generate tests for HW and SW: The solutions of the underlying constraint satisfaction problem (CSP) need to be uniformly distributed over the space of all feasible solutions. This allows the test engineers to perform less tests while maintaining reasonable test coverage.

Uniform sampling from the solution space of a CSP is #P-hard. This means that realistic algorithms need to resort to approximations. We have reviewed several recent approaches for sampling classical CSPs: Random walk strategies for sampling SAT ([WES04]), bucket elimination method ([DKBE02]) and Monte Carlo sampling method which uses probabilistic inference algorithm called *Iterative Join-Graph Propagation* (IJGP, [GD06]).

Our work extends these previous results to constraint satisfaction problems *with preferences* – SoftCSP. We have defined a probability distribution over the solution space of SoftCSP problems. In such distribution, all solutions which satisfy all hard constraints have probability greater than zero, and the probability of optimal solutions is exponentially proportional to that of worse solutions.

We have used the IJGP algorithm to sample from this distribution. We have modified IJGP to use generalized arc-consistency to speed up the computation of probability by removing values which cannot satisfy all hard constraints from variable domains.

The constraint satisfaction problems which occur in test generation can have domains of size up to 2^{64}. IJGP turns out to be very slow on problems with domains larger than binary. To improve IJGP run-time performance on problems with moderate domain size, we have proposed a new version of IJGP called *Interval-IJGP*. This version works only on problems with integer domains. It uses intervals of values as an additional representation together with regular domains. The intervals can have different length. By this we ensure that intervals which are the most likely to contain a value which participates in a solution span the smallest number of values. The number of intervals per domain can be parametrized, thus giving the users an option to trade-off accuracy versus speed of search.

We have tested the IJGP-based algorithms together with a Monte Carlo Markov Chain algorithm called *Gibbs sampling*. We have used problems from the SoftCSP benchmark library *CostFunctionLib* ([Sch]) and a relaxed version of an *all-interval-series problem*, which was proposed by Intel as a useful example of a SoftCSP problem encountered in the HW test generation practice.

Gibbs sampling had the best performance overall in the tests. It was by far the fastest algorithm and the samples it generated corresponded well with the target distribution. Its samples had also the largest variance, which makes this algorithm especially useful for test generation. However, Gibbs samples had significant troubles to satisfy problems with many hard constraints.

IJGP-based algorithms were very slow on most of the problems. Moreover, they did not reflect the target distribution well. As we expected, Interval-IJGP was faster than pure IJGP while its accuracy was similar to IJGP.

We have concentrated only on a subset of issues that arise in CSP solution sampling. Further research could focus on

- Sampling from solution space of problems specified as constraint hierarchies. They would be especially useful for the HW testing community, because they allow an easier description of the problem than weighted CSP.

- Providing more theoretical justification on belief propagation algorithms, and Interval-IJGP in particular

- Investigating other possible domain representations for larger domains, such as sets of binary masks (used by Bin *et al.* in [BESZ02]) and decision trees

Bibliography

[Ake78] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

[BESZ02] Eyal Bin, Roy Emek, Gil Shurek, and Avi Ziv. Using a Constraint Satisfaction Formulation and Solution Techniques for Random Test Program Generation. *IBM Systems journal*, 41(3):386–402, 2002.

[BFBW92] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Higher-Order and Symbolic Computation*, 5(3):223–270, 1992.

[BMR97] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based Constraint Satisfaction and Optimization. *Journal of ACM*, 44(2):201–236, March 1997.

[BR97] C. Bessiere and J.C. Regin. Arc consistency for general constraint networks: Preliminary results. *Proceedings IJCAI 97*, 1:398–404, 1997.

[BW58] W.J. Baumol and P. Wolfe. A Warehouse-Location Problem. *Operations Research*, 6(2):252–263, 1958.

[Chv] Václav Chvátal. Colouring the queen graphs. `http://users.encs.concordia.ca/~chvatal/queengraphs.html`.

[Coo90] Gregory F. Cooper. The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks. *Artificial Intelligence*, 42(2-3):393–405, March 1990.

[Dec96] Rina Dechter. Bucket Elimination: A Unifying Framework for Probabilistic Inference. In E. Horvitz and F. Jensen, editors, *Twelthth Conference on Uncertainty in Artificial Intelligence*, pages 211–219, Portland, Oregon, 1996.

[Dec97a] Rina Dechter. Bucket Elimination: A Unifying Framework for Processing Hard and Soft Constraints. *Constraints*, 2(1):51–55, April 1997.

[Dec97b] Rina Dechter. Mini-Buckets: A General Scheme for Generating Approximations in Automated Reasoning. In *IJCAI*, pages 1297–1303, 1997.

[Dec99] Rina Dechter. Bucket Elimination: A Unifying Framework for Reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[Dec03] Rina Dechter. *Constraint Processing (The Morgan Kaufmann Series in Artificial Intelligence).* Morgan Kaufmann, May 2003.

[DH03] Václav Dupač and Marie Hušková. *Pravděpodobnost a matematická statistika.* Univerzita Karlova v Praze, Praha, 2003.

[DKBE02] R. Dechter, K. Kask, E. Bin, and R. Emek. Generating random solutions for constraint satisfaction problems. *Eighteenth National Conference on Artificial Intelligence*, pages 15–21, 2002.

[DKM02] Rina Dechter, Kalev Kask, and Robert Mateescu. Iterative Join-Graph Propagation. In *Proceedings of the 18th Annual Conference on Uncertainty in Artificial Intelligence (UAI-02)*, pages 128–13, San Francisco, CA, 2002. Morgan Kaufmann.

[DLR77] AP Dempster, NM Laird, and DB Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[DO91] Richard A. Demillo and Jefferson A. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, September 1991.

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, July 1960.

[Fre97] Eugene C. Freuder. In Pursuit of the Holy Grail. *Constraints*, 2(1):57–61, April 1997.

[GD06] V. Gogate and R. Dechter. A New Algorithm for Sampling CSP Solutions Uniformly at Random. In *Principles and Practice of Constraint Programming*, volume LNCS 4204, pages 711–715. Springer Verlag, May 2006.

[GG84] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE transactions on pattern analysis and machine intelligence*, 6(6):721–741, 1984.

[Gil95] W. R. Gilks. *Markov Chain Monte Carlo in Practice*. Chapman & Hall/CRC, December 1995.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, January 1979.

[Has70] W. K. Hastings. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika*, 57(1):97–109, 1970.

[Hoo00] H.H. Hoos. SATLIB: An Online Resource for Research on SAT. *Sat2000: Highlights of Satisfiability Research in the Year 2000*, pages 283–292, 2000.

[Jer03] Mark Jerrum. *Counting, Sampling and Integrating: Algorithms and Complexity (Lectures in Mathematics. ETH Zürich)*. Birkhäuser Basel, April 2003.

[JVV86] MR Jerrum, LG Valiant, and VV Vazirani. Random generation of combinatorial structures from a uniform. *Theoretical Computer Science*, 43(2-3):169–188, 1986.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[KS80] R. Kindermann and J.L. Snell. *Markov random fields and their applications*. American Mathematical Society Providence, RI, 1980.

[LFL$^+$95] D. Lewin, L. Fournier, M. Levinger, E. Roytman, and G. Shurek. Constraint Satisfaction for Test Program Generation. *Computers and Communications, 1995. Conference Proceedings of the 1995*

IEEE Fourteenth Annual International Phoenix Conference on, pages 45–48, 1995.

[LKD01] J. Larrosa, K. Kask, and R. Dechter. Up and Down Mini-Bucket: A Scheme for Approximating Combinatorial Optimization Tasks. Technical report, University of California, Irvine, 2001.

[Mac77] A. K. Mackworth. On Reading Sketch Maps. Technical report, University of British Columbia, Vancouver, British Columbia, Canada, 1977.

[McM04] McMinn, P. . Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[MWJ99] Kevin Murphy, Yair Weiss, and Michael Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 467–47, San Francisco, CA, 1999. Morgan Kaufmann.

[NSZ07] Amir Nahir, Yossi Shiloach, and Avi Ziv. Using Linear Programming Techniques for Scheduling-Based Random Test-Case Generation. *Hardware and Software, Verification and Testing*, pages 16–33, 2007.

[Ott] Lars Otten. WCSP File Format Specification. `http://graphmod.ics.uci.edu/group/WCSP_file_format`.

[Pea86] J. Pearl. Fusion, Propagation, and Structuring in Belief Networks. *Artificial Intelligence*, 29(3):241–288, September 1986.

[Pea87] J Pearl. Evidential reasoning using stochastic simulation of causal models. *Artificial Intelligence*, 32(2):245–257, 1987.

[Pea88] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[Sch] Thomas Schiex. SoftCSP Benchmark Library. `http://mulcyber.toulouse.inra.fr/gf/project/costfunctionlib/scmsvn/`.

[SFV95] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. *Proc. IJCAI95*, 726, 1995.

[SH81] LG Shapiro and RM Haralick. Structural Descriptions and Inexact Matching. *IEEE trans. pattern analy. and mach. intellig.*, 3(5):504–519, 1981.

[SJ89] Alistair Sinclair and Mark Jerrum. Approximate Counting, Uniform Generation and Rapidly Mixing Markov Chains. *Information and Computation*, 82(1):93–133, July 1989.

[SLM92] Bart Selman, Hector J. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[Vel] Miroslav Velev. Miroslav Velev's SAT Benchmarks. `http://www.miroslav-velev.com/sat_benchmarks.html`.

[WES04] W. Wei, J. Erenrich, and B. Selman. Towards Efficient Sampling: Exploiting Random Walk Strategies. *Proceedings of AAAI*, 4:670–676, 2004.

[YFW00] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Generalized belief propagation. In *NIPS*, pages 689–695, 2000.

[YFW03] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Understanding Belief Propagation and its Generalizations. *Exploring Artificial Intelligence in the New Millennium*, pages 239–269, 2003.

[YSP$^+$99] Jun Yuan, Kurt Shultz, Carl Pixley, Hillel Miller, and Adnan Aziz. Modeling Design Constraints and Biasing in Simulation Using BDDs. In *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 584–590, Piscataway, NJ, USA, 1999. IEEE Press.