Algorithms for Stochastic Constraint Satisfaction Problems

BALAFOUTIS ATHANASSIOS

A THESIS SUBMITTED
FOR THE MASTER DEGREE

DEPARTMENT OF
INFORMATION & COMMUNICATION
SYSTEMS ENGINEERING

UNIVERSITY OF AEGEAN

2006

Acknowledgment

A journey is easier when you travel together. Interdependence is certainly more valuable than independence. I would like to thank my teachers here in the Department of Information & Communication Systems Engineering of Aegean University. I owe them an immense debt of gratitude for having me shown a high quality way of research. It is a pleasant aspect that I have now the opportunity to express my gratitude for all of them.

First of all I would like to thank my direct supervisor, Lecturer Kostas Stergiou. He was the person who introduced me to the topic of Stochastic Constraint Programming. The motivating discussions with him play an essential role in the preparation of this thesis. He also taught me how to sharpen ideas and present ideas precisely and clearly. Kostas has patiently read and commented on every draft of this thesis. His overly enthusiasm and integral view on research and his mission for providing 'only high-quality work and not less', has made a deep impression on me. I am lucky to have him as the supervisor of my research.

I would also like to thank my parents for the support they have provided me all the time. I am also very grateful for my wife Christina, whose love, patience and support are always the source of strength, and to our daughter Maria who provided an additional and joyful dimension to my life.

Contents

Chapter 1

Introduction

1.1 Context

The constraint satisfaction problem (CSP) [Mack92], [Dech92], framework has been proposed as a generic way of modeling discrete constrained decision problems, for which generic deduction and search algorithms can be defined. The framework, as well as the associated algorithms, assume that all the components of the instance to consider (variables, domains of possible values, constraints to satisfy) are completely known before modeling and solving it and do not change either during or after modeling and solving.

However, it has been observed for a long time that such assumptions do not hold in many situations, especially when one has to deal with uncertain and dynamic environments.

One of the difficulties of problem modeling and solving in uncertain and dynamic environments comes from the fact that, on the one hand, the knowledge about the real world is often incomplete, imprecise and uncertain and, on the other hand, the real world and the knowledge about it may change during or after modeling and solving.

For example, to be more concrete, consider a travel management system (TMS), embedded in a car, in charge of the management of all the features of a long travel: route, stops, reservations, rendezvous, car refueling and maintenance, etc. The physical system is the car. The user is the car driver. Its physical environment is the road, the traffic, the

weather, etc. Other entities are hotels, restaurants, garages, other people and similar TMSs. Uncertainty may come from the car (actual state of its components), from the environment (actual state of the road, future traffic and weather), and from the other entities (actual availability). Changes may occur at any time from the driver herself (changes in her goals or in her current plan), from the car (unexpected breakdowns), from the environment (traffic jams), and from the other entities (unavailability, rendezvous cancellations).

In order to handle uncertainties and changes in constraint solving (like the TMS problem described above), a number of different modeling frameworks have been proposed. The research work reported in this thesis, considers one of these frameworks; the stochastic constraint satisfaction problem (SCSP) proposed in [Walsh02]. In this framework, we include in the problem definition the available knowledge about possible changes from the real world. The SCSP framework is inspired from the stochastic satisfiability problem (SSAT) [Littm01].

In SCSPs, variables are divided into controllable ones (decision variables) and uncontrollable ones (state variables). State variables, also called stochastic variables, follow a probability distribution. The expressional power of the SCSP can be help us model situations where there are probabilistic estimations about various uncertain events, such as stock market prices, energy demands, weather conditions, etc. SCSPs have very recently been introduced and only a few solution methods have been proposed.

In this thesis, we give the semantics for stochastic constraint programming, we present the existing complete algorithms presented in bibliography and we propose advanced solution methods for SCSPs. Finally, we compare the introduced algorithms with existing ones via a set of experiments.

1.2 Organization of the thesis

This thesis consists of six chapters. Chapter1 includes a general introduction in the area of interest. The related work that has been done in CSPs and SCSPs is reviewed in Chapter 2. We also describe here the main algorithms that have been proposed for solving stochastic constraint satisfaction problems.

In Chapter 3 we propose a generalized arc consistency (GAC) algorithm for SCSPs. This algorithm extends the GAC algorithm AC2001/3.1 with specialized features, so that SCSPs can be handled. We also explain how arc consistency reasoning can be performed when “chance” constraints are present in a problem.

In Chapter 4 we introduce new search algorithms for solving stochastic constraint satisfaction problems. We first identify and correct a flaw in the forward checking (FC) algorithm given in [Walsh02]. We also describe an improved version of FC which exploits probabilities in a more "global" way and in this way results in stronger pruning. Then we introduce a Maintaining Arc Consistency (MAC) algorithm for SCSPs. In contrast with [Walsh02], where the given algorithms can only handle binary constraints, our MAC algorithm is able to handle constraints of any arity. The chapter ends with the presentation of some heuristics which increase the efficiency of the above search algorithms.

A set of experiments is presented in Chapter 5. These experiments demonstrate the effect that the flaw has in the FC algorithm of [Walsh02] and depict the achieved improvement of our new FC algorithm. We also present experiments with FC that uses arc consistency as a preprocessing technique.

Finally Chapter 6 concludes the thesis by summarizing the results reported here and gives some directions for future work.

Chapter 2

Related Work

2.1 Constraint Satisfaction Problems

A wide variety of problems in Artificial Intelligence, Engineering, Databases, and other disciplines can be modeled as Constraint Satisfaction Problems (CSPs). Constraints are a declarative knowledge representation formalism that allows for a compact and expressive modeling of many real life problems. In the next few paragraphs we present some formal definitions about classical constraint satisfaction problems, and we introduce the basic techniques for processing and solving CSPs.

2.1.1 Basic definitions

A *constraint satisfaction problem* (CSP) consists of a finite set of variables, X_1, X_2, …, X_n each associated with a domain D_1, D_2, …, D_n of possible values, and a set of constraints, C_1, C_2, …, C_m. Each constraint C_i is a relation, defined on some subset of the variables, and specifies the allowed combination of values for that subset. Alternatively, constraints can be described by mathematical expressions or by computable procedures.

For each constraint, the number of variables it constraints is called the *arity* of the constraint and the variables themselves are called the *scope* of the constraint.

Constraints can be either *binary* over pairs of variables, or *non-binary* over any number of variables. For example consider a CSP with *n* variables X_1, X_2, …, X_n each with domain {1, 2,…, n}. The constraint $C_1(X_1, X_2) = \{(0, 0), (0, 2), (2, 3)\}$ is a *binary constraint* between variables X_1 and X_2. The constraint $C_2(X_1, X_2, X_3) = \{(0, 0, 0), (0, 1, 2), (2, 0, 3)\}$ is an example of a non-binary constraint. A binary CSP is one with only binary constraints. A non-binary CSP is one with constraints of any arity. Another simple type of constraint is the *unary constraint*, which restricts the value of a single variable. Every unary constraint can be eliminated simply by pre-processing the domain of the corresponding variable to remove any value that violates the constraint.

A state of the problem is defined by an assignment of values to some or all of the variables. An assignment that does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is mentioned. A solution is an assignment of a value to each variable from its domain such that all the constraints are satisfied. Typical tasks in constraint satisfaction problems are to determine whether a solution exists, to find one or all solutions, and to find an optimal solution relative to a given cost function.

An example of a constraint satisfaction problem is the well known k-colorability problem. The task is to color, if possible, a given graph with k colors only, such that any two adjacent nodes have different colors. A constraint satisfaction formulation of this problem associates the nodes of the graph with variables, the possible colors are their domains and the not-equal constraints between adjacent nodes are the constraints of the problem.

Another known constraint satisfaction problem concerns satisfiability (SAT), which is the task of finding a truth assignment to propositional variables such that a given set of clauses expressed in CNF are satisfied. For example, given the two clauses $(A \vee B \vee \neg C)$, $(\neg A \vee D)$, the assignment of false to A, true to B, false to C, and false to D is a satisfying truth value assignment.

The structure of a constraint problem is usually depicted by a constraint graph or hyper-graph in the case of a non-binary problem, whose nodes represent the variables, and any two nodes are connected if the corresponding variables participate in the same constraint scope. In the k-colorability formulation, the graph to be colored is the constraint graph. In the SAT example above, the constraint graph has A connected with D, and A, B and C are connected to each other.

Constraint problems have proven successful in modeling mundane cognitive tasks such as vision, language comprehension, default reasoning and abduction, as well as in applications such as scheduling, design, diagnosis, and temporal and spatial reasoning. The reason is that constraints allow for a natural, expressive and declarative formulation of what has to be satisfied, without the need to say how it has to be satisfied.

In general, constraint satisfaction tasks (like finding one or all solutions, or the best solution) are computationally intractable (NP-hard). Intuitively, this mean, that in the worst case all the possible variable instantiations may need to be considered before a solution (or best solution) can be found by a complete algorithm. However, there are some tractable classes of problems that allow for efficient solution algorithms even in the worst-case. Moreover, also for non-tractable classes, many techniques exhibit a good performance in practice in the average case.

2.1.2 Complete Techniques for solving CSPs

Given the finiteness of the domain for each variable, it is always possible in principle to find a solution for a CSP if one exists. In order to get an assignment of variables from the respective domains, we simply check all possible assignments exhaustively to see whether there is any assignment satisfying all the constraints simultaneously. This technique is called generate and test in logic programming. The backtracking search technique is an improvement over generate and test.

Backtracking Search

In Backtracking search (BT), variables are instantiated one by one. After each instantiation of a variable, all the constraints involving this variable and already instantiated variables will be checked. If some constraint is not satisfied, we avoid instantiating the rest of the variables because the constraint will still be violated no matter how we instantiate them. In other words, a portion of the search space is pruned. A new value will be chosen for the current variable. If none of the values for the variable satisfy the related constraints, backtracking occurs. The algorithm goes back to the previous variable and chooses a new value for it. The process will be repeated until a solution is found or there is no choice of value for the first variable in which case there is no solution. Given a CSP (V,D,C), an illustrative algorithmic schema for the backtracking paradigm is shown below:

```
1.  algorithm BackTracking((V,D,C))
2.  Begin
3.     i ← 0; backtracking ← false;
4.  while i < n do //exists variable not assigned yet
5.       if not backtracking then
6.              Choose a variable v_i from V-{v_0, v_1, ..., v_{i-1}};
7.              S_i ← D_j; //S_i is the current domain
8.       Endif
9.       backtracking ← true;
10.      while S_i is not empty do // search a value for current v_i
11.             choose a value {a} for v_i;
12.             S_i ← S_i – {a};
13.             enforce certain level of consistency on the network;
14.             If the domain of some variable is empty then
15.             restore the domains of variables V-{v_0, v_1, ..., v_{i-1}};
16.             else //{a} is a valid value for v_i
17.             backtracking ← false;
18.             break;
19.             Endif
20.      Endwhile
21.      if backtracking then
22.             i ← i - 1; // backtrack to the previous variable
23.             If i < 0 then break; endif
24.      else i ← i + 1; // progress to the next variable
25. Endwhile
26. if backtracking then report unsatisfiability;
27. else report the solution;
28. End
```

To implement BT based on the algorithm described above, we must check in line 13, if the current value {a} of variable v_i, satisfies all the constraints with previous variables.

Forward Checking Search

One way to make better use of constraints during search is called Forward Checking (FC). The main idea in this algorithm is to reduce domains of unassigned variables based on assigned variables. Each time a variable is instantiated, we delete from the domains of the uninstantiated variables all of those values that conflict with the current variable assignment. Whenever a variable X is assigned, the Forward Checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y 's domain any value that is inconsistent with the value chosen for X. The main advantage of this algorithm is that identifies dead ends without having to try them via backtracking. To implement FC we can use the procedure for BT. The FC algorithm comes up, if we substitute line 13 with a call to the following function:

```
Function UPDATE (unlab_vars,doms,cons,current_var_assignment)
//returns an updated set of domains
   for each variable y in unlab_vars do
      for each value v in Dy'  do
           if(y,v)  is  incompatible  with  current_var_assignment
           with  respect  to  the  constraint  between  y  and  the
           current variable
        then Dy' ← Dy' – {v}
       end
   end
   return doms'
```

Arc Consistency (AC)

Although forward checking detects many inconsistencies, it does not detect all of them. It does not look far enough ahead in the search tree. *Constraint propagation* is the general term for propagating the implications of a constraint on one variable onto other variables. The idea of *arc consistency* provides a fast and efficient method of constraint propagation that is substantially stronger than FC. The key idea for this technique is to delete values from a variable's domain if these values are not supported by any value in some constraint. Thus, applying arc consistency, at each step of the search, results in early detection of an inconsistency that is not detected by pure forward checking. We can also apply AC as a pre-processing step before we start search. In that way we can reduce

the size of the search tree and in some cases we can discover inconsistent problems. We now give a more formal definition of arc consistency

Given a constraint network (V, D, C), the *support* of a value $a \in D_i$ under a constraint c_{ij} is a value $b \in D_j$ such that (a, b) satisfies constraint c_{ij}. The value a is *viable* with respect to c_{ij} if it has a support in D_j. A constraint c_{ij} is consistent along the arc (x_i, x_j), if and only if every $a \in D_i$ has a support in D_j. A constraint c_{ij} is arc consistent if and only if it is consistent along both arcs (x_i, x_j) and (x_j, x_i), A constraint network is arc consistent if and only if every constraint in the network is arc consistent.

AC-3 [Mac77a, McG79] is a generic popular arc consistency algorithm. To enforce arc consistency in a constraint network, a key task of AC-3 is to check the viability of a value with respect to any related constraint. REVISE-3(x_i, x_j) is a function that removes those values in D_i without any support in D_j under c_{ij}. If any value in D_i is removed when revising (x_i, x_j), all binary constraints (or arcs) pointing to x_i, except c_{ij}, will be revised. A queue Q is used to hold these arcs for later processing. It can be shown that this algorithm is correct. The AC-3 algorithm is shown below:

```
1.  Algorithm AC-3
2.     Begin
3.        Q ← {(xi, xj)| cij∈C or cji∈C, i≠j}
4.        while Q not empty do
5.           select and delete any arc (xi, xj) from Q
6.           if REVISE-3(xi, xj) then
7.              Q ← Q ∪ {(xk, xi)| cki∈C, k≠j}
8.     End

1.  Procedure REVICE-3(xi, xj)
2.     Begin
3.        DELETE ← false
4.        for each a ∈ Di do
5.           if there is no b ∈ Dj such that cij(a,b) then
6.              delete a from Di
7.              DELETE ← true
8.        return DELETE
9.     End
```

Maintaining Arc Consistency (MAC)

MAC [Dan94] is a backtracking search scheme (see Figure 1) for finding a solution of a constraint network. Under this scheme, the network is pre-processed by an AC algorithm. During search, arc consistency is maintained (i.e. enforced) after each instantiation of a variable in order to prune the search space. This process of maintaining arc consistency is denoted by *mac*; in the figure, V denotes the most recent assigned variable. Obviously, a key component of *mac* is AC.

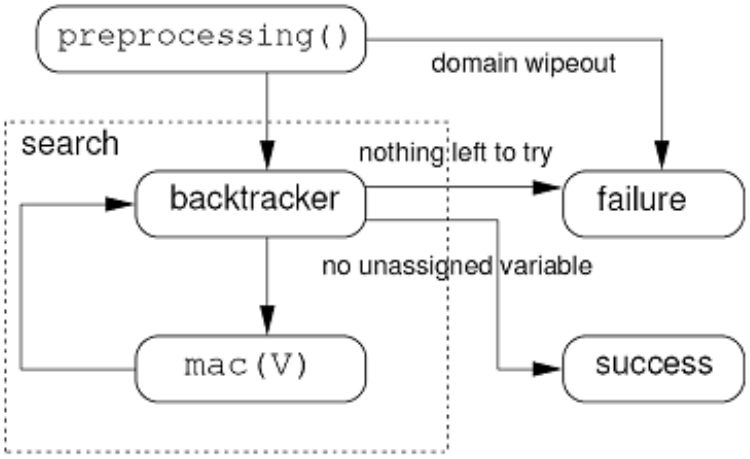


Figure 1. MAC Schema

Given an AC algorithm, it is easy to design a MAC algorithm based on it. For instance, MAC3 is derived from AC3 by embedding AC3 in backtracking search. To implement MAC we can use the procedure for BT. The MAC algorithm comes up, if we substitute line 13 with a call to the AC-3 algorithm.

2.2 Constraint Solving in Uncertain and Dynamic Environments

In classic CSPs, the associated algorithms assume that all the components of the instance to consider (variables, domains of possible values, constraints to satisfy) are completely known before modeling and solving it and do not change either during or after modeling and solving.

However, it has been observed for a long time that such assumptions do not hold in many situations, specifically each time one has to deal with uncertain and dynamic environments.

One of the difficulties of problem modeling and solving in uncertain and dynamic environments comes from the fact that, on the one hand, the knowledge about the real world is often incomplete, imprecise and uncertain and, on the other hand, the real world and the knowledge about it may change during or after modeling and solving.

In order to handle uncertainties and changes in constraint solving, a number of different modeling frameworks have been proposed. In the next paragraphs we review some of them.

2.2.1 Dynamic constraint satisfaction problem (DCSP)

The DCSP was proposed in [Mitt90] and is defined as a sequence of CSPs, each one derived from some changes in the definition of the previous one. These changes may affect any component in the problem definition: variables (additions or removals), domains (changes in the intensional definitions, value additions or removals in case of extensional definitions), constraints (additions or removals), constraint scopes (variable additions or removals), or constraint definitions (changes in the intensional definition, tuple additions or removals in case of extensional definition). Because domains can be seen as unary constraints, because variables are implicitly added or removed with all the constraints that apply to them, and because any change in a component can be seen as a removal followed by an addition, all these changes can be basically expressed in terms of constraint additions or removals.

2.2.2 Conditional constraint satisfaction problem (CCSP)

The basic objective of the CCSP framework [Sabin98] is to model problems whose solutions do not all have the same structure, i.e. do not all involve the same set of variables and constraints. Such a situation occurs when dealing with product configuration or design problems, because the physical systems that can meet a set of user requirements do not all involve the same components. More generally, it occurs when dealing with any synthesis problem, such as design, configuration, planning, scheduling, etc. In a CCSP, the set of variables is divided into a set of mandatory variables and a set of optional ones. The set of constraints is also divided into a set of compatibility constraints and a set of activity constraints. Compatibility constraints are classical constraints. Activity constraints define the conditions of activation of the

optional variables as a function of the current assignment of other mandatory or optional variables.

Constraints are activated only if their variables are activated too. When solving a CCSP, the structure of the problem (activated variables and constraints) may change as a function of the current assignment. Thus, a CCSP can be considered as a particular case of DCSP where all the possible changes are defined by the activity constraints. However, the methods that have been proposed so far for dealing with DCSP and with CCSP are very different from each other.

2.2.3 Open constraint satisfaction problem (OCSP)

In an OCSP, the allowed values in domains, as well as the allowed tuples in relations, may not be all known when starting a search for a solution [Falt02]. They may be acquired online when no solution has been found with the currently known values and tuples. Such a situation occurs each time the acquisition of information about domains and relations is a costly process that needs heavy computation or requests to distant sites. Thus, an OCSP can be considered as a particular case of DCSP where all the possible changes result in extension of the domains and relations.

2.2.4 Mixed constraint satisfaction problem (MCSP)

The MCSP was proposed in [Farg96] to model decision problems under uncertainty about the actual state of the real word. In an MCSP, variables are divided into controllable ones (decision variables) that are under the control of the decisional agent and uncontrollable variables (state variables) that are not under its control. In such a framework, a basic request may be to build a decision (an assignment of the decision variables) that is consistent whatever the state of the world (the assignment of the state variables) is.

The model of the possible changes takes the form of uncontrollable variables, besides usual controllable ones, which may take any value in their domains. A usual objective is to produce a solution that is valid whatever the values taken by uncontrollable variables. As an example, let us consider the small instance in Figure 2 and let us assume that variables *x, y* and *t* are controllable (we can decide upon their value), but that variable *z* is uncontrollable (it may take any value: 1 or 2). An arc connecting two values denotes that

they are compatible. In such conditions, $\{x = 1, y = 3, t = 3\}$ is a solution because it is consistent with any value of z, but $\{x = 1, y = 2, t = 2\}$ is not because it is not consistent with $z = 2$.

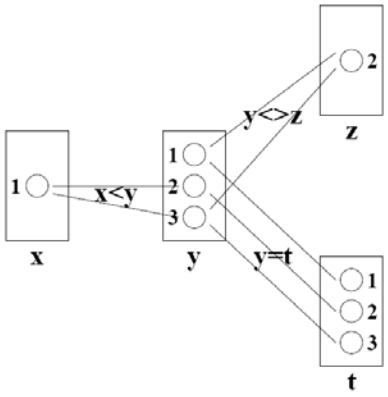


Figure 2

2.2.5 Probabilistic constraint satisfaction problem (PCSP)

The PCSP was proposed in [Farg93] to model decision problems under uncertainty about the presence of constraints. In a PCSP, a probability of the presence in the real world is associated with each constraint. In such a framework, a basic request may be to produce an assignment that maximizes its probability of consistency in the real world. A PCSP is a particular case of the valued constraint satisfaction problem (VSCP).

The model of the possible changes takes the form of a probability of existence associated with each constraint. A usual objective is, to produce a solution whose probability of validity is maximum. As an example, let us consider the small instance in Figure 3, slightly different from the one in Figure 2 (one less value in z's and t's domains) and let us assume the following independent probabilities of existence associated with each constraint: $P(x < y) = 0.2$, $P(y \neq z) = 1$, $P(y = t) = 0.6$ (the constraint $y \neq z$ is certain; the other ones are uncertain).In these conditions, some solutions are certainly not valid, such as $\{x = 1, y = 2, z = 2, t = 1\}$ (constraint $y \neq z$ violated). Other ones have a non null probability of validity such as $\{x = 1, y = 1, z = 2, t = 2\}$ (constraint $x < y$ and $y = t$ violated; $P = (1 – 0.2)(1 – 0.6) = 0.32$). But, the one whose probability of validity is maximum is $\{x = 1, y = 1, z = 2, t = 1\}$ (constraint $x < y$ violated; $P = 1 – 0.2 = 0.8$).

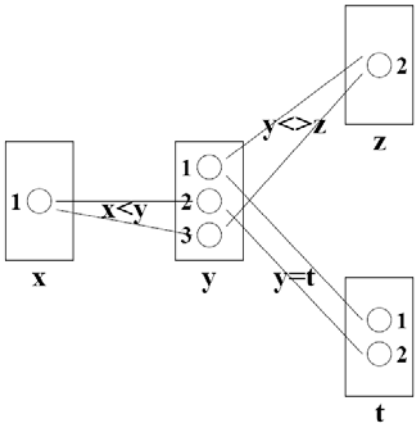


Figure 3

2.2.6 Stochastic constraint satisfaction problem (SCSP)

The SCSP framework was proposed in [Walsh02] to model decision problems under uncertainty about the actual state of the real world, in the same way as the MSCP framework. The SCSP framework is inspired from the stochastic satisfiability problem (SSAT). As in an MCSP, variables in a SCSP are divided into controllable ones (decision variables) and uncontrollable ones (state variables). The main difference from a MCSP is that a probability distribution is associated with the domain of each state variable. Another difference is that the requests can freely alternate state and decision variables. In such, a framework, a basic request may be, as in the PCSP framework, to build a decision (an assignment of the decision variables) that maximizes the probability of consistency in the real world.

In this model, a probability distribution is associated with the domain of each uncontrollable variable. A usual objective is thus to produce a solution whose probability of validity is maximum. As an example, let us consider the same instance in Figure 2, but let us assume now that variables *x, z,* and *t* are controllable, but the variable *y* is uncontrollable. In such conditions, there is no solution that is valid whatever the value taken by *y*. However, let us assume the following probability distribution over *y*'s domain: P(y = 1) = 0.1, P(y = 2) = 0.7, P(y = 3) = 0.2. In these conditions, some solutions are certainly not valid, such as, {x = 1, z = 1, t = 1} (inconsistency whatever the value taken by y). Other ones have a non null probability of validity such as {x = 1, z = 2, t = 3} (consistency if y = 3; P = 0.2). But, the one whose probability of validity is maximum is {x = 1, z = 1, t = 2} (consistency if y = 2; P = 0.7).

2.2.7 Branching constraint satisfaction problem (BCSP)

The BCSP was proposed in [Fowl00] to model sequential decision problems under uncertainty about the arrival of new elements (objects, tasks…). In a BCSP, at each step, present variables are assigned when possible, taking into account variables that will be added to the problem definition. A utility is associated with each assigned variable. At each step, a probability of addition is associated with each absent variable. The request is, at each step, to assign the added variable with a value that maximizes the global expected utility.

The model of possible changes takes the form of a probability of addition to the current problem associated with each possible additional variable. Moreover, each variable may be assigned or not and a gain is associated with its assignment. The objective is to produce an assignment of the current variables that maximises the expected value of its extension to the additional variables. As an example, let us consider the same instance in Figure 3 and let us assume that variables y and z are present in the current problem, but that variables x and t are not. They may be added, x with a probability of 0.8 and t with a probability of 0.6. Moreover, let us assume that the gains associated with the assignment of variables x, y, z and t are respectively equal to 10, 20, 5, and 10. In such conditions, the expected gain associated for example with the assignment $\{y = 1, z = 2\}$ is equal to $20 + 5 + 0.6\ 10 = 31$, because it will be possible to assign t, but not possible to assign x. In fact, the optimal assignment is $\{y = 2\}$ (z not assigned) with an expected gain equal to $20 + 0.8\ 10 + 0.6\ 10 = 34$, because it will be possible to assign both x and t.

2.3 Stochastic Constraint Programming

In this Section we will review with more details the work that has been done in the SCSP framework, which is the main subject of this thesis. As mentioned, many decision problems contain uncertainty. Data about events in the past may not be known exactly due to errors in measuring or difficulties in sampling, whilst data about events in the future may simply not be known with certainty. For example, when scheduling power stations, we need to cope with uncertainty in future energy demands. As a second example, nurse rostering in an accident and emergency department requires us to anticipate variability in workload. As a final example, when constructing a balanced bond portfolio, we must deal with uncertainty in the future price of bonds. To deal with such

situations, [Walsh02] proposed an extension of constraint programming called *stochastic constraint programming* in which we distinguish between decision variables, which we are free to set, and stochastic (or observed) variables, which follow some probability distribution.

2.3.1 Stochastic constraint programs

The simplest possible model is a one-stage stochastic constraint satisfaction problem (stochastic CSP) in which the decision variables are set before the stochastic variables are given values. This models situations in which we must act now and observe later. For example, we have to decide now which nurses to have on duty and will only later discover the actual workload. We can easily invert the instantiation order if the application demands, with the stochastic variables set before the decision variables. Constraints are defined (as in traditional constraint satisfaction) by relations of allowed tuples of values. Constraints can, however, be implemented with specialized and efficient algorithms for consistency checking. The stochastic variables independently take values with probabilities given by a probability distribution.

A one stage stochastic CSP is satisfiable iff there exists values for the decision variables so that, given random values for the stochastic variables, the probability that all the constraints are satisfied equals or exceeds a threshold θ. The probabilistic satisfaction of constraints allows us to ignore worlds (values for the stochastic variables) which are rare. Note that the definition reduces to that of a traditional constraint satisfaction problem if we have no stochastic variables and $\theta = 1$.

In a two stage stochastic CSP, there are two sets of decision variables, V_{d1} and V_{d2}, and two sets of stochastic variables, V_{s1} and V_{s2}. The aim is to find values for the variables in V_{d1}, so that given random values for V_{s1}, we can find values for V_{d2}, so that given random values for V_{s2}, the probability that all the constraints are satisfied equals or exceeds θ. Note that the values chosen for the second set of decision variables V_{d2} are conditioned on both the values chosen for the first set of decision variables V_{d1} and on the random values given to the first set of stochastic variables V_{s1}. This can model situations in which items are produced and can be consumed or put in stock for later consumption. Future production then depends both on previous production (earlier decision variables) and on previous demand (earlier stochastic variables). An m stage stochastic CSP is defined in an analogous way to one and two stage stochastic CSPs.

A stochastic constraint optimization problem (stochastic COP) is a stochastic CSP plus a cost function defined over the decision and stochastic variables. The aim is to find a solution that satisfies the stochastic CSP which minimizes (or, if desired, maximizes) the expected value of the cost function.

2.3.2 Production planning example

The following stochastic constraint program taken from [Walsh02], models a simple m quarter production planning problem. In each quarter, we will sell between 100 and 105 copies of a book. To keep customers happy, we want to satisfy demand in all m quarters with 80% probability. At the start of each quarter, we decide how many books to print for that quarter. This problem is modelled by an m stage stochastic CSP. There are m decision variables, x_i representing production in each quarter. There are also m stochastic variables, y_i representing demand in each quarter. These take values between 100 and 105 with equal probability. There is a constraint to ensure 1st quarter production meets 1st quarter demand:

$$x_1 \geq y_1$$

There is also a constraint to ensure 2nd quarter production meets 2nd quarter demand plus ant unsatisfied demand or less any stock:

$$x_2 \geq y_2 + (y_1 - x_1)$$

And there is a constraint to ensure j^{th} quarter production ($j \geq 2$) meets j^{th} quarter demand plus any unsatisfied demand or less any stock:

$$x_j \geq y_j + \sum_{i=1}^{j-1} (y_i - x_i)$$

We must satisfy these m constraints with a threshold probability $\theta = 0.8$. This stochastic CSP has a number of solutions including $x_i = 105$ for each i. (i.e. always produce as many books as the maximum demand). However, this solution will tend to produce books surplus to demand which is undesirable.

Suppose storing surplus book costs \$1 per quarter. We can define an m stage stochastic COP based on this stochastic CSP in which we additionally minimize the expected cost of storing surplus books:

$$\sum_{j=1}^{m} \min\left(\sum_{i=1}^{j} x_i - y_i, 0 \right)$$

Note that a solution to a stochastic CSP or COP defines how to set later decision variables given the values for earlier stochastic and decision variables.

2.3.3 Semantics

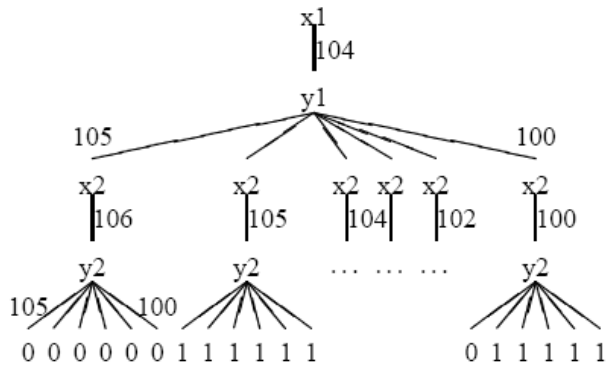
A stochastic constraint satisfaction problem is a 6-tuple (V, S, D, P, C, θ) where V is a list of variables, S is the subset of V which are stochastic variables, D is a mapping from V to domains, C is a set of constraints over V, and θ is a threshold probability in the interval [0, 1]. Constraints are defined by a set of variables and a relation giving the allowed tuples of values. Variables are set in the order in which they appear in V. Thus, in an one stage stochastic CSP, V contains the decision variables and then the stochastic variables. In a two stage stochastic CSP, V contains the first set of decision variables, the first set of stochastic variables, then the second set of decision variables, and finally the second set of stochastic variables.

A *policy* is a tree with nodes labelled with variables, starting with the first variable in V labelling the root, and ending with the last variable in V labelling the nodes directly above leaves. Nodes labelled with decision variables have a single child, whilst nodes labelled with stochastic variables have one child for every possible value. Edges in the tree are labelled with values assigned to the variable labelling the node above. Leaf nodes are labelled with 1 if the assignment of values to variables along the path to the root satisfies all the constraints and 0 otherwise. Each leaf node corresponds to a possible world and has an associated probability; if s_i is the i^{th} stochastic variable on a path to the root, d_i is the value given to s_i on this path, (i.e. the label of the following edge), and $prob(s_i = d_i)$ is the probability that $s_i = d_i$, then the probability of this world is simply:

$$\Pi_i \; prob(s_i = d_i)$$

The *satisfaction* of a policy is defined as the sum of the leaf values weighted by their probabilities. A policy satisfies the constraints iff its satisfaction is at least θ. A stochastic CSP is satisfiable iff there is a policy which satisfies the constraints. The optimal satisfaction of a stochastic CSP is the maximum satisfaction of all policies. For a stochastic COP, the expected value of a policy is the sum of the objective valuations of each leaf node weighted by their probabilities. A policy is optimal if it satisfies the constraints and maximizes (or, if desired, minimizes) the expected value.

Consider again the production planning and a two-quarter policy that sets $x_1 = 104$ and if $y_1 > 100$ then $x_2 = y_1 + 1$, else $y_1 = 100$ and $x_2 = 100$. We can represent this policy by the following (partial) tree:



By definition each of the leaf nodes in this tree is equally probable. There are 6^2 leaf nodes, of which only 7 are labeled 0. Hence, this policy's satisfaction is (36-7)/36, and the policy satisfies the constraints as this just exceeds $\theta = 0.8$.

2.3.4 Complete algorithms

In this section we present the backtracking and forward checking algorithms for solving stochastic CSPs as they appear in [Walsh02]. Note that both algorithms are defined for binary constraints only.

Backtracking

We assume that variables are instantiated in order. However, if decision variables occur together, they can be instantiated in any order. On meeting a decision variable, the backtracking (BT) algorithm tries each value in its domain in turn. The maximum value is returned to the previous recursive call. On meeting a stochastic variable, we try each value in turn, and return the sum of the all answers to the subproblems weighted by the probabilities of their occurrence. At any time, if instantiating a decision or stochastic variable breaks a constraint, we return 0. If we manage to instantiate all the variables without breaking any constraint, we return 1.

```
Procedure BT(i, θl, θh)
if i > n then return 1
θ := 0
q := 1
for each dj ∈ D(xi)
   if θh ∈ S then
      p := prob(xi → dj)
      q := q − p
      if consistent(xi → dj) then
         θ := θ + p × BT( i+1, (θl - θ - q)/p, (θh - θ)/p )
         if θ > θh then return θ
      if θ + q < θl then return θ
   else
      if consistent(xi → dj) then
         θ := max(θ, BT(i+1, max(θ, θl), θh))
         if θ > θh then return θ
 return θ
```

Upper and lower bounds, θ_h and θ_l are used to prune search. By setting $\theta_l = \theta_h = \theta$, we can determine if the optimal satisfaction is at least θ. Alternatively, by setting $\theta_l = 0$ and $\theta_h = 1$, we can determine the optimal satisfaction. The calculation of upper and lower bounds in recursive calls requires some explanation. Suppose that the current assignment to a stochastic variable returns a satisfaction of θ_0. We can safely ignore other values for this stochastic variable if $\theta + p \times \theta_0 \geq \theta_h$. That is, if $\theta_0 \geq \frac{\theta_h - \theta}{p}$. This gives the upper bound in the recursive call to BT on a stochastic variable. Alternatively, we cannot hope to satisfy the constraints adequately if $\theta + p \times \theta_0 + q \leq \theta_l$ as q is the maximum that the remaining values can contribute to the satisfaction. That is, if $\theta_0 \leq \frac{\theta_l - \theta - q}{p}$. This gives the lower bound in the recursive call to BT on a stochastic variable. Finally, suppose that the current assignment to a decision variable returns a satisfaction of θ. If this is more than θ_l, then any other values must exceed θ to be part of a better policy. Hence, we can replace the lower bound in the recursive call to BT on a decision variable by $\max(\theta, \theta_l)$. Because of these bounds, value ordering heuristics can reduce search. For decision

variables, we should choose values that are likely to return the optimal satisfaction. For stochastic variables, we should choose values that are more likely.

To better understand how bounds are updated when a new recursive call to BT is made, and how they are exploited to avoid needless exploration of some parts of the search tree, the reader should note that bounds provide information about the already achieved satisfaction at each point in search and also about the maximum satisfaction that can be achieved thereafter. If the already achieved satisfaction plus the maximum satisfaction that can be achieved by exploring the rest of the problem is less than the desired threshold then the algorithm backtracks to the previously instantiated decision variable and tries another value for it. To put it simply, this means that the currently explored policy cannot achieve the threshold, so a new policy must be explored. Similarly, in the case where we seek the maximum satisfaction, if the currently explored policy cannot offer higher satisfaction than the maximum satisfaction that has been already discovered then the algorithm abandons it and moves on to explore a different policy.

Forward Checking

The Forward Checking (FC) procedure is based on the BT algorithm. On instantiating a decision or stochastic variable, the FC algorithm checks forward and prunes values from the domains of future decision and stochastic variables which break constraints. Checking forwards fails if a stochastic or decision variable has a domain wipeout (dwo), or if a stochastic variable has so many values removed that we cannot hope to satisfy the constraints. As in the regular forward checking algorithm, we can use an 2-dimensional array, $prune(i, j)$ to record the depth at which the value d_j for the variable x_i is removed by forward checking. This is used to restore values on backtracking. In addition, each stochastic variable, x_i has an upper bound, q_i on the probability that the values left in its domain can contribute to a solution. When forward checking removes some value, d_j from x_i, we reduce q_i by $prob(x_i \rightarrow d_j)$, the probability that x_i takes the value d_j. This reduction on q_j is undone on backtracking. If forward checking ever reduces q_i to less than θ_l, we backtrack as it is impossible to set x_i and satisfy the constraints adequately.

```
Procedure FC(i, θl, θh)
   if i > n then return 1
   θ := 0
```

```
   for each d_j ∈ D(x_i)
      if prune(i, j) = 0 then
         if check(x_i → d_j, θ_l) then
            if x_i ∈ S then
               p := prob(x_i → d_j)
               q_i := q_i - p
               θ := θ + p × FC(i+1, (θ_l - θ - q)/p, (θ_h - θ)/p)
               restore(i)
               if θ + q_i < θ_l then return θ
               if θ > θ_h then return θ
            else
               θ := max(θ, FC(i+1, max(θ, θ_l), θ_h))
               restore(i)
               if θ > θ_h then return θ
         else restore(i)
   return θ


Procedure check(x_i → d_j, θ_l)
   for k := i + 1 to n
      dwo := true
      for d_l ∈ D(x_k)
         if prune(k, l) = 0 then
            if inconsistent(x_i → d_j, x_k → d_l) then
               prune(k, l) := i
               if x_k ∈ S then
                  q_k := q_k – prob(x_k → d_l)
                  if q_k < θ_l then return false
            else dwo := false
      if dwo := false return false
   return true

Procedure restore(i)
   for j = i + 1 to n
      for d_k ∈ D(x_j)
         if prune(j, k) = i then
            prune(j, k) = 0
         if x_j ∈ S then q_j := q_j + prob(x_j → d_k)
```

Chapter 3

Generalized Arc Consistency for Stochastic CSPs

In this section we describe a generalized arc consistency algorithm for SCSPs. First we review the AC (and GAC) algorithms for classical CSPs that are used as basis. Then we present the GAC algorithm for stochastic CSPs and introduce a pruning rule that can be used to delete values from certain decision variables. Finally, we discuss how GAC can be enhanced to handle "chance" constraints, an important extension of the SCSP framework.

3.1 AC2001/3.1 for Binary Constraints

The use of constraint propagation is the main feature of any constraint solver. All the constraint solvers use propagation as a basic step. It is thus of prime importance to

manage the propagation in an efficient and effective fashion. Each improvement to a constraint propagation algorithm has an immediate effect on the performance of the constraint solving engine.

The AC-3 algorithm presented in Chapter 2 is a common generic constraint propagation algorithm. Unfortunately, the worst case time complexity of AC-3 is $O(ed^3)$, where e is the number of constraints and d is the size of the maximum domain in a problem. A refinement of AC-3 with better worst case time complexity is the AC2001/3.1 algorithm [Bess05].

AC2001/3.1 is a worst case optimal arc consistency algorithm based on AC-3. It preserves the simplicity of AC-3 while improving on AC-3 in efficiency both in terms of constraint checks and in terms of cpu time. The worst case time complexity of AC2001/3.1 is $O(ed^2)$, with space complexity $O(ed)$. In the next section we will present AC2001/3.1 with more details.

The worst case time complexity of AC-3 is based on a naive implementation of line 5 of the REVISE-3 procedure described in Chapter 2 (b is always searched from scratch). However, from the analysis we know a constraint (x_i, x_j) may be revised many times. The key idea to improve the efficiency of the algorithm is that we need to find from scratch a support for a value $a \in D_i$ only in the first revision of the arc (x_i, x_j), and store the support in a structure $Last((x_i, a), x_j)$. When checking the viability of $a \in D_i$ in the subsequent revisions of the arc (x_i, x_j), we only need to check whether its stored support $Last((x_i, a), x_j)$ is still in the domain D_j. If it was removed (because of the revision of other constraints), we would just have to explore the values in D_j that are "after" the support since its "predecessors" have already been checked before.

Assume without loss of generality that each domain D_i is associated with a total ordering $<_d$. The function $succ(a, D_j)$, where D_j denotes the current domain of x_j during the procedure of arc consistency enforcing, returns the first value in D_j that is after a in accordance with $<_d$, or NIL, if no such an element exists. We define NIL as a value not belonging to any domain but preceding any value in any domain.

As a simple example, let the constraint c_{ij} be $x_i = x_j$, with $D_i = D_j = [1..11]$. The removal of value 11 from D_j (say, after the revision of some arc leaving x_j) leads to a revision of (x_i, x_j). REVISE-3 will look for a support for every value in D_i , for a total cost of $1 + 2 + \ldots + 9 + 10 + 10 = 65$ constraint checks, whereas only $(x_i, 11)$ had lost support. The REVISE2001/3.1 procedure listed below, makes sure that for each $a \in [1..10]$, $Last((x_i, a), x_j)$ still belongs to D_j, and finds that $Last((x_i, 11), x_j)$ has been removed. Looking for a new support for 11 does not need any constraint check since D_j does not

contain any value greater than $Last((x_i, 11), x_j)$, which was equal to 11. It saves 65 constraint checks compared to AC-3.

AC2001/3.1 is the main algorithm AC-3 augmented with the initialization of $Last((x_i, a), x_j)$ to be *NIL* for any constraint c_{ij} and any value $a \in D_i$. The corresponding revision procedure, REVISE2001/3.1 is listed below:

```
Procedure REVISE2001/3.1(x_i, x_j)
 begin
   DELETE ← false
   for each a ∈ D_j do
      b ← Last((x_i, a), x_j)
      if b ∉ D_j then
         b ← succ(b, D_j)
         while (b ≠ NILL) and (¬ c_ij(a, b)) do
            b ← succ(b, D_j)
         if b ≠ NILL then
            Last((x_i, a), x_j) ← b
         else
            delete a from D_i
            DELETE ← true
   return DELETE
 end
```

3.2 GAC2001/3.1 for Non-binary Constraints

AC2001/3.1 can be extended to GAC2001/3.1 to deal with non-binary constraints. The definition of arc consistency for non binary constraints is a direct extension of the binary one [Mac77a]. Let us denote by $var(c_j) = (x_{j1}, \ldots, x_{jq})$ the sequence of variables involved in a constraint c_j, by $rel(c_j)$ the set of tuples allowed by c_j, and by $D^{var(c_j)}_{|x_i=\alpha}$ the set of the tuples τ in $D_{j_1} \times \ldots \times D_{j_q}$ with $\tau[x_i] = \alpha$ (where $i \in \{j_1, \ldots, j_q\}$).

A tuple τ in $D^{var(c_j)}_{|x_i=\alpha} \cap rel(c_j)$ is called a support for (x_i, α) on c_j. The constraint c_j is *arc consistent* (also called *generalized arc consistent*, or *GAC*) iff for any variable x_i in $var(c_j)$, every value $\alpha \in D_i$ has a support on c_j. Tuples in a constraint c_j are totally

ordered with respect to the lexicographic ordering obtained by combining the ordering $<_d$ of each domain with the ordering of the sequence $var(c_j)$ (or with respect to any total order used when searching for support). Once this ordering is defined, a call to REVISE2001/3.1 (x_i, c_j) checks for each $\alpha \in D_i$ whether $Last((x_i, a), c_j)$, which is the smallest support found previously for (x_i, a), still belongs to $D^{\text{var}(c_j)}$. If not, it looks for a new support for α on c_j. If such a support τ exists, it is stored as $Last((x_i, a), c_j)$, otherwise α is removed from D_i. The function $succ\left(\tau, D^{\text{var}(c_j)}_{|x_i=\alpha}\right)$ returns the smallest tuple in $D^{\text{var}(c_j)}_{|x_i=\alpha}$ greater than τ.

```
1.   Algorithm GAC2001/3.1(x, c)
2.      begin
3.         Q ← {(x_i, c_j) | c_j ∈ C, x_i ∈ var(c_j)}
4.         while Q not empty do
5.            select and delete any pair (x_i, c_j) from Q
6.            if REVISE2001/3.1 (x_i, c_j) then
7.               Q ← Q ∪ {(x_k, c_m) | c_m ∈ C, x_i, x_k ∈ var(c_m), m ≠ j, i ≠ k}
8.      End
```

```
Procedure REVISE2001/3.1(x_i, c_j)
   begin
   DELETE ← false
   for each α ∈ D_i do
      τ ← Last((x_i, a), c_j)
      if ∃ k/τ[x_{j_k}] ∉ D_{j_k} then
         τ ← succ(τ, D^{var(c_j)}_{|x_i=α})
         while (τ ≠ NIL) and (¬c_j(τ)) do
            τ ← succ(τ, D^{var(c_j)}_{|x_i=α})
         if τ ≠ NIL then
            Last((x_i, a), c_j) ← τ
         else
            delete α from D_i
            DELETE ← true
   return DELETE
end
```

3.3 Specialized features for stochastic CSP

The idea of Arc Consistency can also be applied to SCSPs as a pre-processing step, before we start search. In that way we can reduce the size of the search tree and in some cases we can discover inconsistent problems.

The AC2001/3.1 (for binary constraints) and GAC2001/3.1 (for non-binary constraints) algorithms presented above, can directly be used for SCSPs. But, as they have been designed for standard CSPs, they do not exploit the special characteristics of SCSPs. Thus, their direct use is inefficient.

In the next paragraphs we extent the AC2001/3.1 and GAC2001/3.1 algorithms with specialized features, so that SCSPs can be handled efficiently.

3.3.1 AC2001/3.1 for Stochastic CSPs

To improve the efficiency of AC2001/3.1 on SCSPs we start from the following observation. Whenever a value $\alpha \in D(x_i)$ of a stochastic variable $x_i \in S$ is deleted (because it has no support in some other variable) then the maximum threshold θ_{max} that can be obtained is reduced. To compute the reduced threshold θ_{max} we can add the probabilities of the remaining values of variable x_i, $sum(P(x_i))$, do the same for all other stochastic variables, and then multiply all sums for all stochastic variables. If θ_{max} falls under the desired threshold θ then we can infer that the problem is inconsistent. That is, the required satisfaction cannot be achieved.

As an example, let us consider a small instance of a SCSP. Let us assume that x is a decision variable and y, z are stochastic variables. We suppose that all variables have the same domain $D(0, 1, 2)$, $\theta = 0.7$ and that the probability distribution for variables y, z are as follows: $P(y = 0) = 0.5$, $P(y = 1) = 0.3$, $P(y = 2) = 0.2$, $P(z = 0) = 0.2$, $P(z = 1) = 0.6$ and $P(z = 2) = 0.2$. We also define two constraints: $C_1(x, y) = \{(0, 0), (0, 1), (1, 1), (2, 1)\}$ and $C_2(x, z) = \{(0, 1), (0, 2), (1, 1), (2, 2)\}$. By checking C_1, the value $y = 2$ has no support and an AC algorithm will erase that value. The maximum threshold that can now be obtained is computed as: $\theta_{max} = (0.5 + 0.3) \times (0.2 + 0.6 + 0.2) = 0.8$. Because $\theta_{max} > \theta$, we can continue by checking C_2. The value $z = 0$ has no support and an AC algorithm will erase that value. The new maximum threshold that can be obtained is computed as: $\theta_{max} = (0.5 + 0.3) \times (0.6 + 0.2) = 0.64$. Because $\theta_{max} < \theta$, we can infer that the problem is inconsistent.

3.3.2 GAC2001/3.1 for Stochastic CSPs

Arc consistency for SCSPs be generalized to deal with non-binary constraints. Whenever a value $\alpha \in D(x_i)$ of a stochastic variable $x_i \in S$ is deleted (because it has no support tulpe τ in some other constraint then the maximum threshold θ_{max} that can be obtained is reduced. To compute the reduced threshold θ_{max} we can add the probabilities of the remaining values of variable x_i, $sum(P(x_i))$, do the same for all other stochastic variables, and then multiply all sums for all stochastic variables. If θ_{max} falls under the threshold θ then we can infer that the problem is inconsistent.

The extended Stochastic GAC2001/3.1 (SGAC2001/3.1) and the corresponding procedure (SREVISE2001/3.1) for stochastic Generalized Arc Consistency are listed below:

```
Algorithm SAC2001/3.1(x, c)
   begin
      Q ← {(x_i, x_j)| c_ij∈C or c_ji∈C, i≠j}
      while Q not empty do
         select and delete any pair (x_i, c_j) from Q
         if SREVISE2001/3.1(x_i, c_j) = 1 then
            Q ← Q∪{(x_k, c_m)|c_m∈C,x_i,x_k∈var(c_m),m≠j, i≠k}
         else if SREVISE2001/3.1(x_i, c_j) = -1 then
            problem inconsistent; Exit;
   End


Procedure SREVISE2001/3.1(x_i, c_j)
   Begin
   DELETE ← false
   for each α ∈ D_i do
      τ ← Last((x_i, a), c_j)
      if ∃k/τ[x_{j_k}] ∉ D_{j_k} then
         τ ← succ(τ, D^{var(c_j)}_{|x_i=α})
         while (τ ≠ NIL) and (¬c_j(τ)) do
            τ ← succ(τ, D^{var(c_j)}_{|x_i=α})
         if τ ≠ NIL then
            Last((x_i, a), c_j) ← τ
```

```
            Else
                delete α from D_i
                if x_i ∈ S then
                    θ_max = sum(P(x_i))
                    for each x ∈ S do
                        θ_max(x) = sum(P(x))
                        θ_max ×= θ_max(x)
                    End
                    if θ_max < θ then
                        ACTION ← -1
                    Else
                        ACTION ← 1
    return ACTION
End
```

Note that the described method of determining whether a problem is satisfiable when applying (G)AC is meaningful as a preprocessing technique only when we try to determine if the satisfaction of the problem is at least as much as a given threshold θ. In the case where we seek the maximum satisfaction, the condition described above will never be met (i.e. θ_{max} will never fall below $\theta_l = 0$ unless there is domain wipeout of a stochastic variable). This means that stochastic (G)AC will work just like standard (G)AC. However, if (G)AC is maintained during search then the above algorithm can be used to prune the search space in both cases.

3.3.3 A Pruning Rule for Stochastic GAC

In the previous subsections we showed how (G)AC can exploit probabilities to determine, in some cases, that the problem cannot be satisfied. A natural question that arises is whether AC and GAC can actually perform value pruning in a SCSP, apart from the standard case where a value has no support in some constraint. We will now describe a pruning rule that can be used by an AC or GAC algorithm to prune values from certain decision variables. First we will explain how the rule can be applied in binary problems and then we will generalize to the non-binary case. For simplicity reasons, we will describe how the pruning rule can be used when preprocessing a SCSP. The application of the rule during search is feasible, but more involved, and is left as future work.

As we will see through this section, this pruning rule can be applied only in problems in which the first stage consists of decision variables (i.e. one or more decision variables are in front of any stochastic variable in the sequence of variables). Note that this restriction concerns only the application of the rule as a preprocessing technique.

Example 3.1

We assume that x is a decision variable and y, z are stochastic variables. We suppose that all variables have the same domain $D(0, 1, 2)$, $\theta = 0.7$ and that the probability distribution for variables y, z are as follows: $P(y = 0) = 0.2$, $P(y = 1) = 0.1$, $P(y = 2) = 0.7$, $P(z = 0) = 0.6$, $P(z = 1) = 0.3$ and $P(z = 2) = 0.1$. We also define two constraints: $C_1(x, y)$: Disallowed tuples $\{(0, 0), (0, 1)\}$ and $C_2(x, z)$: Disallowed tuple$(0, 1)$.

Whenever we try to find a support for value 0 of the decision variable x, the SGAC2001/3.1 algorithm will find a support value 2 for the stochastic variable y and the support values $\{0, 2\}$ for the stochastic variable z. Since value $\{0\}$ of the decision variable x has support in both other variables, it will not be erased by the AC algorithm. But if we try to solve this problem using a search algorithm, we will find out that the assignment $x = 0$, throws the threshold under θ ($\theta_{max} = 0.7 \times 0.7 = 0.49 < \theta$). Therefore, the assignment $x = 0$ cannot participate in a solution and should not be considered.

To increase the pruning efficiency, an AC algorithm could check if the supports of each value of x in variables y, z offer enough satisfaction to achieve the desired threshold θ. If not, as is the case with value 0 in the example 3.1, then the corresponding value can safely be deleted from the domain of x. Note that this would not be the case is x belonged to a later stage of the problem (i.e. it was after some stochastic variables). In the example, when AC looks for supports for value 0 of x, instead of only locating supports in y and z, it can also check if the maximum satisfaction θ_{max} that the supports of 0 can offer is less that the desired threshold. In this case: $\theta_{max} = 0.7 \times (0.6 + 0.1) = 0.47 < \theta$. Therefore value 0 of x will be deleted.

More generally, a pruning rule that can be checked every time we are looking for supports for a value α of a decision variable x (when this variable is present in the first stage of the problem) can be expressed as: "For all stochastic variables add the probabilities of the values that are supported by $x=a$ and then multiply all sums. If the result is lower than the threshold then remove value α from the domain of variable x".

3.3.4 Chance Constraints for Stochastic SCSPs

As already defined, a stochastic constraint satisfaction problem is a 6-tuple (V, S, D, P, C, θ) where V is a list of variables, S is the subset of V which are stochastic variables, D is a mapping from V to domains. C is a set of constraints where a constraint $c \in C$ on variables x_i, …, x_j specifies a subset of the Cartesian product $D(x_i) \times \ldots \times D(x_j)$ indicating mutually-compatible variable assignment.

In [Walsh03], an extension of the stochastic CSP is presented, where the definition of *chance constraints* is introduced. Any constraint of C that involves at least one variable in S is called a *chance constraint*, h. Every chance constraint h has an associated probability threshold θ_h in the interval [0, 1] indicating the minimum probability with which the chance constraint h must be satisfied. That is, the minimum fraction of worlds where the constraint is satisfied. Constraints that involve only decision variables, or have $\theta_h = 1$ are hard constraints, meaning that must always be satisfied.

Note that [Walsh02] only allowed for one (global) chance constraint so the definition of stochastic constraint programming that allows for multiple chance constraints is strictly more general.

Let us denote by $con(x_i) = \{c_{i1}, \ldots, c_{iq}\}$ the set of constraints in which the stochastic variable x_i participates and by $\theta_h(c_i)$ the threshold for the chance constraint c_i. Then every time a value $\alpha \in D(x_i)$ is deleted, we must check if the maximum threshold that can be obtained in all constraints that belong to $con(x_i)$ is lower than $\theta_h(c_i)$. This could be done by adding the probabilities of the values in all stochastic variables $\in c_i$ and then multiplying all sums.

The extension of SREVISE2001/3.1 to support chance constraints is listed below:

```
Procedure SREVISE2001/3.1(x_i, c_j)
   Begin
   DELETE ← false
   for each α ∈ D_i do
      τ ← Last((x_i, a), c_j)
      if ∃ k/τ[x_{j_k}] ∉ D_{j_k} then
         τ ← succ(τ, D^{var(c_j)}_{|x_i=α})
         while (τ ≠ NIL) and (¬c_j(τ)) do
            τ ← succ(τ, D^{var(c_j)}_{|x_i=α})
```

```
        if τ ≠ NIL then
            Last((x_i, a), c_j) ← τ
        Else
            delete α from D_i
            if x_i ∈ S then
                θ_max(x_i) = sum(P(x_i))
                for each c_i ∈ con(x_i) do
                    for each x ∈ c_i do
                        θ_max(x) = sum(P(x))
                        θ_hmax(c_i) ×= θ_max(x)
                if θ_hmax(c_i) < θ_h(c_i) then
                    ACTION ← -1
                for each x ∈ S do
                    θ_max ×= θ_max(x)
                if θ_max < θ then
                    ACTION ← -1
                Else
                    ACTION ← 1
            End
    return ACTION
End
```

During search the chance constraints can be handled as follows. Every time a value is deleted from a stochastic variable, we first detect all constraints in which this variable participates. Then we check in any constraint that is a chance constraint if the remaining probabilities are enough to achieve the threshold of the chance constraint. This could be done by adding the probabilities of the values in all stochastic variables that participate on the current constraint and then multiplying all sums.

Chapter 4

Search Algorithms

In this chapter we introduce new search algorithms for solving stochastic constraint satisfaction problems. We first identify and correct a flaw in the forward checking (FC) algorithm given in [Walsh02]. We also describe an improved version of FC which exploits probabilities in a more "global" way and in this way results in stronger pruning. Then we introduce a Maintaining Arc Consistency (MAC) algorithm for SCSPs.

Before getting into details on the new search algorithms we recall and analyze the behavior of the FC algorithm given in [Walsh02].

4.1 Walsh' s FC Algorithm

We recall here again the FC algorithm presented in [Walsh02] in order to examine closely the way the algorithm operates:

```
Procedure FC(i, θ_l, θ_h)
   if i > n then return 1
   θ := 0
```

```
   for each d_j ∈ D(x_i)
      if prune(i, j) = 0 then
         if check(x_i → d_j, θ_l) then
            if x_i ∈ S then
               p := prob(x_i → d_j)
               q_i := q_i - p
               θ := θ + p × FC(i+1, (θ_l - θ - q)/p, (θ_h - θ)/p)
               restore(i)
               if θ + q_i < θ_l then return θ
               if θ > θ_h then return θ
            else
               θ := max(θ, FC(i+1, max(θ, θ_l), θ_h))
               restore(i)
               if θ > θ_h then return θ
         else restore(i)
   return θ
```

The *check* and *restore* procedures are given bellow.

```
     Procedure check(x_i → d_j, θ_l)
1.      for k := i + 1 to n
2.         dwo := true
3.         for d_l ∈ D(x_k)
4.            if prune(k, l) = 0 then
5.               if inconsistent(x_i → d_j, x_k → d_l) then
6.                  prune(k, l) := i
7.                  if x_k ∈ S then
8.                     q_k := q_k – prob(x_k → d_l)
9.                     if q_k < θ_l then return false
10.              else dwo := false
11.        if dwo := false then return false
12.     return true

Procedure restore(i)
   for j = i + 1 to n
      for d_k ∈ D(x_j)
         if prune(j, k) = i then
```

```
    prune(j, k) = 0
if x_j ∈ S then q_j := q_j + prob(x_j → d_k)
```

We will focus our attention on the *check* procedure, which is responsible for making forward checks and deciding whether the current variable assignment should be accepted or rejected. Checking forwards fails if a stochastic or decision variable has a domain wipeout (dwo), or if a stochastic variable has so many values removed that we cannot hope to satisfy the threshold. This last condition is shown in lines 7-9 of *check* procedure. If the variable we want to check is stochastic (x_k) and its value d_1 is inconsistent, we reduce q_k by $prob(x_k \rightarrow d_1)$, the probability that x_k takes the value d_1. If forward checking ever reduces q_k to less than θ_1, we backtrack as it is impossible to set x_k and satisfy the constraints adequately (line 9 in *check* procedure). Below we show that this process may be problematic in some cases.

4.2 Improved Forward Checking

If we examine the Forward Checking algorithm described above in more detail, we will see that it makes some weak assumptions that may even result in returning erroneous results. We will try to reveal the weaknesses of this FC algorithm in the next paragraphs, through a set of examples.

As mentioned, when we are trying to solve a SCSP, there are two different things that we may be looking for. We are looking either for the optimal satisfaction or we are trying to determine if the optimal satisfaction is at least θ (where θ is the threshold). According to [Walsh02] the optimal satisfaction can be achieved by setting $\theta_l = 0$ and $\theta_h = 1$. To determine if the optimal satisfaction is at least θ we can set $\theta_l = \theta_h = \theta$.

In the following two examples we will see that there are some cases where the optimal satisfaction in the FC algorithm (presented by [Walsh02]) cannot be achieved due to a flaw in the check procedure of the algorithm and specifically due to the condition checked in lines 7-9.

Example 4.1

Suppose we have a SCSP with two stochastic variables S1 and S2. The domains of these variables are {0, 1} and the desired threshold is $\theta_l = \theta_h = 0.92$. The probabilities of

the stochastic variables are shown in Figure 4.1. We also have a constraint involving variables S1 and S2 with one disallowed tuple:

Constraint (S1,S2) – Disallowed tuple (1,0)

A graphical representation of the problem's search tree is shown in Figure 4.1.

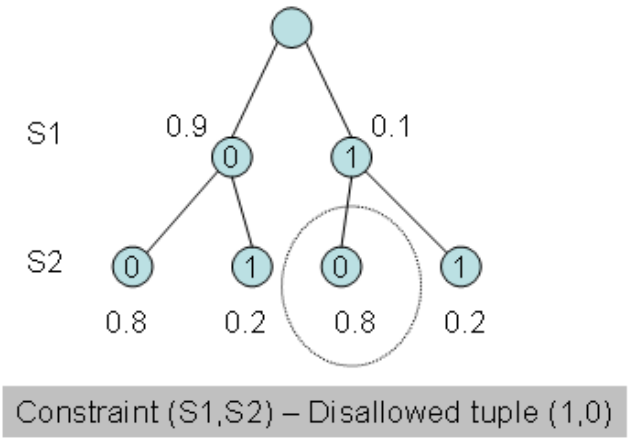


Figure 4.1

During search the algorithm first instantiates variable S1 to 0 and forward checks this assignment. Since it does not violate any constraint and forward checking does not remove any value from variable S2, the algorithm continues by setting S2=0 and then S2=1. After backtracking to S1, the current satisfaction θ for variable S1 will be 0.9. This satisfaction is less that θ_h ($0.9 \times 0.8 = 0.72 < 0.92$), so we will continue by setting S1=1.

When the value 1 is assigned to variable S1, the value 0 will be rejected from the domain of variable S2 because it violates the constraint. Procedure *check* will now determine if the remaining values in the domain of S2 are enough to achieve the threshold. Let's examine this in more detail.

In line 8 (q_k := q_k - prob(x_k → d_1), q_2 will be calculated as follows:

$$q_2 = 1 - prob(S2 \rightarrow 0) = 1 - 0.8 = 0.2$$

In line 9 (if $q_k < \theta_1$ then return false) we will have:

$$q_2 < \theta_1 \rightarrow 0.2 < 0.92, \text{ which is true}$$

Thus, the procedure will return false and the algorithm will be terminated with returned optimal satisfaction 0.9. However, this value is not correct. It is easy to determine from Figure 4.1, that the optimal satisfaction for this problem is 0.92.

Example 4.2

We have 3 variables D1, S1, S2, where D1 is a decision variable and S1, S2 are stochastic. The domains of these variables are {0, 1}. The probabilities of the stochastic variables are shown in Figure 4.2.We also have 2 constraints in this problem:

Constraint (D1,S1) – Disallowed tuple (0,1)
Constraint (S1,S2) – Disallowed tuple (1,0)

In this problem we are looking for the optimal satisfaction, which means that $\theta_l = 0$ and $\theta_h = 1$. A graphical representation of this problem's search tree is shown in Figure 4.2.

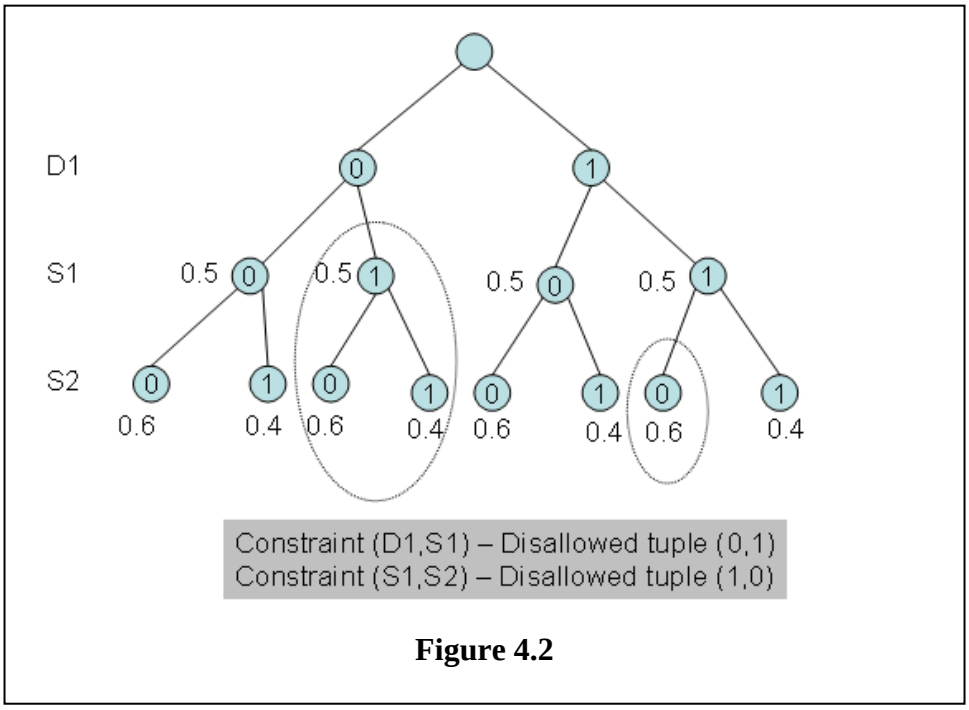


Figure 4.2

The FC algorithm will first instantiate D1 to 0 and forward check this assignment. As a result, value 1 of variable S1 will be deleted and the subtree having this value as root will be pruned. Then the algorithm will explore the non-pruned subtree below D1=0 and eventually will backtrack to D1. At this point θ will be 0.5 (i.e the satisfaction of the explored subtree). Now when FC moves forward to instantiate S1, θ_l will be set to $\max(\theta_l, \theta) = \max(0.5, 0) = 0.5$. The subtree below S1=0, weighted by prob(S1=0), gives 0.5 satisfaction. When assigning 1 to D1, *check* will return false. This is because value 0

of S2 will be removed and the remaining probability in the domain of S2 will be $0.4 < \theta_l$. Therefore, FC will backtrack and terminate, incorrectly returning 0.5 as the maximum satisfaction. Clearly, the maximum satisfaction, which is achieved by this policy, is 0.7.

The last example demonstrates that the pruning achieved by Walsh's FC is weak.

Example 4.3

Let's consider a SCSP with 3 variables D1, S1, S2, where D1 is a decision variable and S1, S2 are stochastic. The domains of these variables are {0, 1} and the requested threshold is 0.7. The probabilities of the stochastic variables are shown in Figure 4.3.We also have 2 constraints in this problem:

Constraint (D1,S1) – Disallowed tuple (0,0)
Constraint (D1,S2) – Disallowed tuple (0,1)

A graphical representation of this problem's search tree is shown in Figure 4.3.

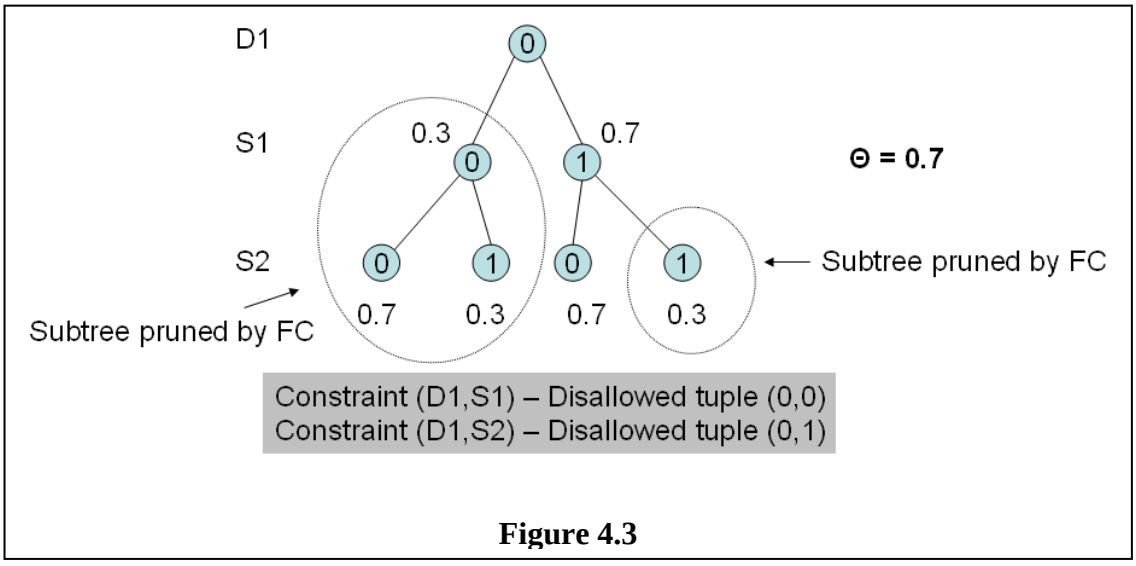


Figure 4.3

When value 0 is assigned to variable D1, the FC algorithm presented in [Walsh02] will first examine variable S1 and it will remove value 0 from its domain since it violates the constraint between D1 and S1. Then the algorithm will check if the remaining value 1 is adequate to achieve the threshold. This check will return true and the algorithm will continue by forward checking the assignment D1=0 against the values of variable S2. In a

similar way as above, value 1 will be removed from the domain of S2 but since the remaining value 0 is adequate to achieve the threshold, FC will continue. The algorithm will now assign value 1 to variable S1 and then value 0 to variable S2. After this last assignment the algorithm will discover that the threshold cannot be achieved ($\theta = 0.7 \times 0.7 = 0.49$) and it will backtrack.

An improved version of the above FC algorithm could discover early that the assignment of value 0 to variable D1 cannot achieve the desired threshold without requiring the instantiation of variables S1 and S2. This can be done while forward checking the assignment of 0 to variable D1 against the future variables. To be precise, this can be easily done by multiplying the probabilities of the remaining values of the future stochastic variables S1, S2 (after forward checking) and comparing the result with threshold.

$$0.7 \times 0.7 = 0.49 < 0.7$$

By summarizing the results of the above examples, we can see in example 3 that the algorithm visits redundant nodes. Moreover, in examples 1 and 2 the algorithm returns a wrong value for the optimal satisfaction. In example 1 this flaw happens when the current variable is stochastic.

The main reason for these failures is the condition in line 9 of the *check* procedure:

$$\texttt{if } q_k < \theta_l \texttt{ then return false}$$

The algorithm considers only the sum q_k of the remaining probabilities of each future stochastic variable on its own. This value is compared with θ_l. This restricts the algorithm to a "local" view of the future problem.

An improved version of the above *check* procedure will not check the q_k quantity with θ_l, but it will compare the θ_l with following quantity:

$$\zeta_\kappa \times \texttt{prob}(x_k \rightarrow d_l) + \theta + q_k \text{ (if the current variable is stochastic)}$$

or

$$\zeta_\kappa \text{ (if the current variable is decision)}$$

where:

ζ_κ : is the product of probabilities of the remaining values of future stochastic variables

$\texttt{prob}(x_k \rightarrow d_l)$: is the probability of current value of variable x_k

θ: is the sum of satisfactions of previous values of the current variable,

q_k: is the sum of the probabilities of the rest values in the current stochastic variable.

This quantity considers value removals from multiple future stochastic variables together. It describes the maximum possible satisfaction of an assignment of a value to

the current variable. If this quantity is less than θ_l then the *check* procedure will return false. This is because it is possible that enough values are removed from a number of stochastic variables so that the maximum possible satisfaction of the current assignment cannot exceed the previously computed satisfaction at the current variable.

The pseudocode of the above described *improved check* procedure is given below:

```
     Procedure improvedCheck(x_i → d_j, θ, θ_l)
1.      for k := i + 1 to n
2.         dwo := true
3.         for d_l ∈ D(x_k)
4.            if prune(k, l) = 0 then
5.               if inconsistent(x_i → d_j, x_k → d_l) then
6.                  prune(k, l) := i
7.                  ζ_i := multiplication[prob(future x_i ∈ S)]
8.                  if x_k ∈ S then
9.                     q_k := q_k - prob(x_k → d_l)
10.                    if x_i ∈ S then
11.                       if (ζ_i × prob(x_k → d_l) + θ + q_k) < θ_l then
12.                          return false
13.                    else  // if variable is decision
14.                       if ζ_i < θ_l then
15.                          return false
16.              else dwo := false
17.         if dwo := false return false
18.     return true
```

By substituting the *check* procedure presented in [Walsh02] with the *improvedCheck* procedure presented here, we generate an improved FC algorithm which returns the correct optimal satisfaction in any case.

Value Ordering

By default, the FC algorithm presented above, simply selects the next unassigned value from each variable, in the order given by the list of values for every variable. This static value ordering seldom results in the most efficient search, in cases we want to determine if the optimal satisfaction is at least θ.

The order of values of the stochastic variables might increase the efficiency of the FC algorithm. This is because every value of a stochastic variable participates in the problem with a different probability. And if we examine first the most probable values of each stochastic variable, it might be faster to reach the threshold θ, or to figure out that threshold θ cannot be reached.

Thus, a simple heuristic that can be used in cases where we want to determine if the optimal satisfaction is at least θ, is the descending order of values of the stochastic variables, according to their probability.

On the other hand, the values of decision variables can be ordered according to any value ordering heuristic used in standard CSPs modified to suit the SCSP case. That is, the values that are most likely to lead to a solution (optimal satisfaction) should be tried first. This can be determined by taking into account the number of supports that each value has in the future variables and the probabilities of the supports of future stochastic variables.

A detailed investigation of value ordering heuristics for stochastic and decision variables is left as future work.

4.3 Maintaining Arc Consistency algorithm

The MAC algorithm presented in Chapter 2 works for classic CSPs. Having defined the AC and FC algorithms for Stochastic CSPs it is easy to extend the classic MAC to SCSPs. The new algorithm can be derived if we substitute the *improvedCheck* procedure of the FC algorithm with a call to the GAC algorithm for Stochastic CSPs presented in Chapter 3. Note that this, provide us with an algorithm that can operate in SCSPs with constraints of any arity.

The MAC algorithm may be more efficient than FC because it can determine inconsistencies in future variables earlier than FC. As an example, let's consider the following SCSP (Figure 4.4):

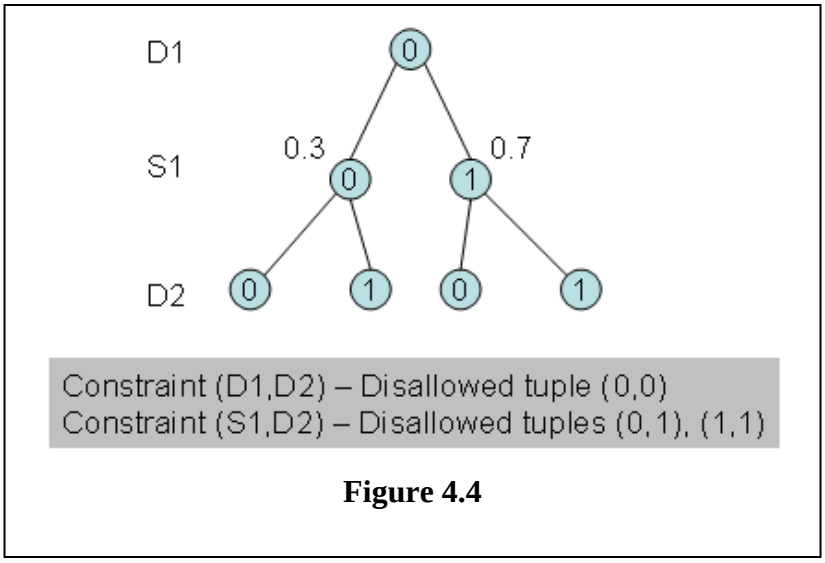


Figure 4.4

When the MAC algorithm sets D1=0 and tries to check the arc consistency of future variables it will see that in variable S1 we have a domain wipeout. The FC algorithm will understand this, after setting S1=0.

In the above example the difference is due to classic propagation as in standard CSPs. As we will see in the next example there are cases where the probabilities of the stochastic variables could play an important role:

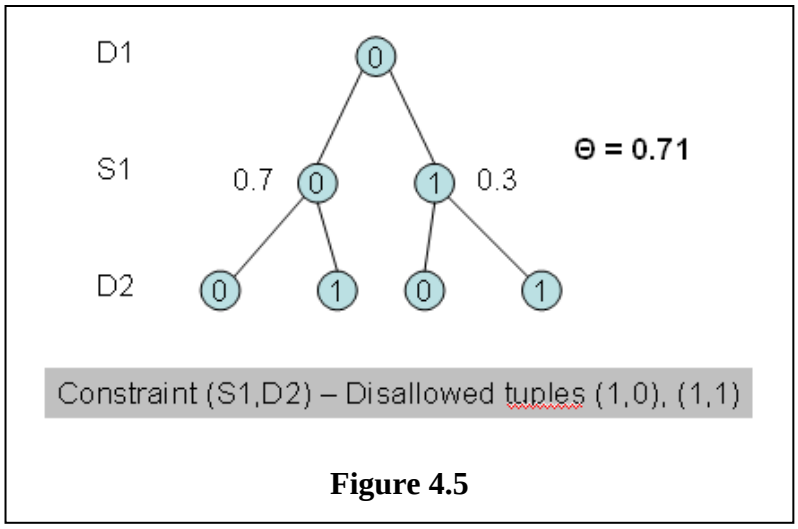


Figure 4.5

In this example (Figure 4.5), when the MAC algorithm sets D1=0 and tries to check the arc consistency of future variables it will see that the remaining values in variable S1

are not enough to achieve the threshold (value S1=1 will be rejected as inconsistent with variable D2). The FC algorithm will understand this, after setting S1=1.

From the above examples it is clear that in some cases the MAC algorithm can visit fewer nodes than FC, in the search tree.

Chapter 5

Experiments

We implemented GAC, FC and Improved FC in Java and ran them on a range of randomly generated problems. In this chapter we summarize the results of our experiments. Note that the experiments presented are preliminary. Further experiments with random and, mainly, realistic problems are required to better evaluate the performance and scalability of the algorithms.

5.1 Problems description

The experiments we ran in this thesis were set up as follows. The problems have 10 variables, each one with 5 values. The variables alternate from decision to stochastic one by one. 100 random problems are generated at each value of p (the percentage of constraints present in a problem out of the n(n-1)/2=45 possible constraints) and q (the percentage of allowed tuples per constraint in a problem out of the d^2=25 possible tuples) from 0 to 1 in steps of 0.1. All the constraints in these problems were binary.

First we ran on these randomly generated problems Walsh's FC algorithm to get an indication of how often the flaw in the algorithm affects the results. Then we ran the improved FC algorithm and compared it with improved FC that uses AC as a preprocessing step.

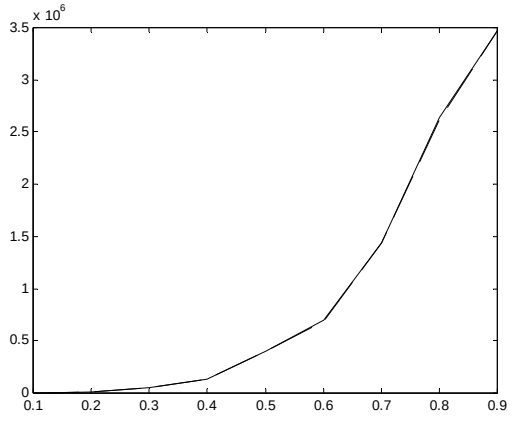
In these problems we measured the search cost (in terms of visited nodes in the explored search tree) and the cpu time in order to find the optimal satisfaction. The search cost and the cpu time for the decision problem of determining if a policy exists to meet a given fixed threshold θ displays a similar (but slightly lower) complexity peak.

5.2 Search Cost plots

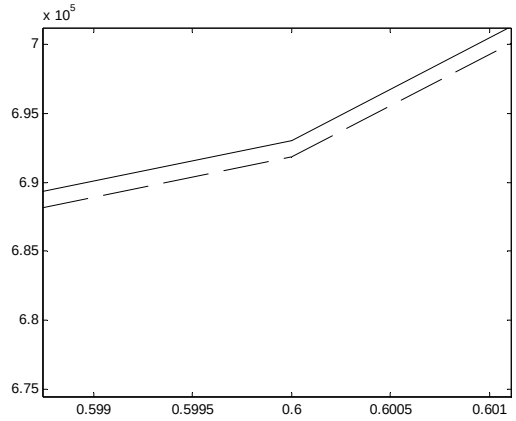
In the diagrams presented in this section, we have on x-axis the percentage of allowed tuples per constraint *q* and on the y-axis the nodes visited by the algorithms.

We present 9 plots each one for every value of *p* from 0.1 to 0.9.That is, the diagrams correspond to problems with increasing density. With the continuous line we represent the improved FC algorithm. With the dashed line we represent the AC plus FC algorithm. Since the curves in all plots are very close, we present also a zoom plot.

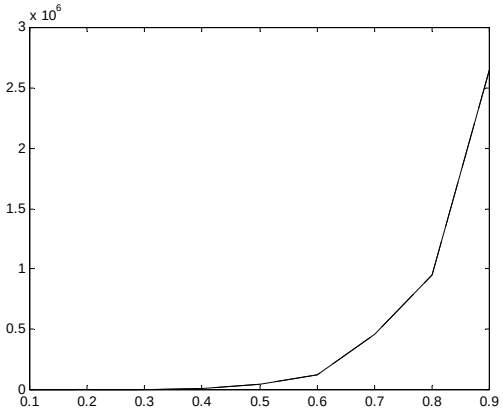
p=0.1



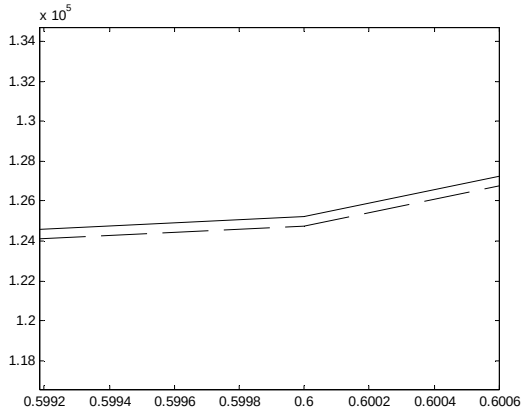
p=0.1 (zoom)



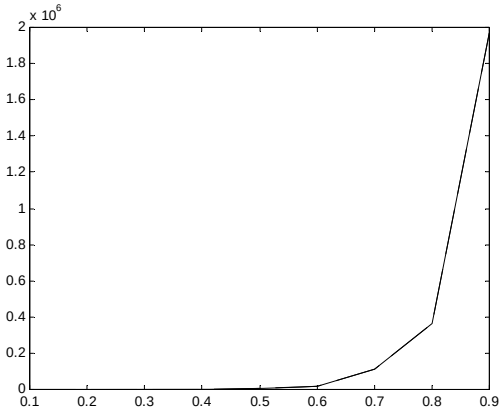
p=0.2



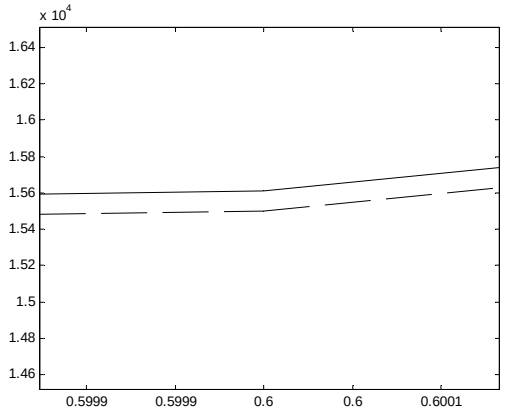
p=0.2 (zoom)



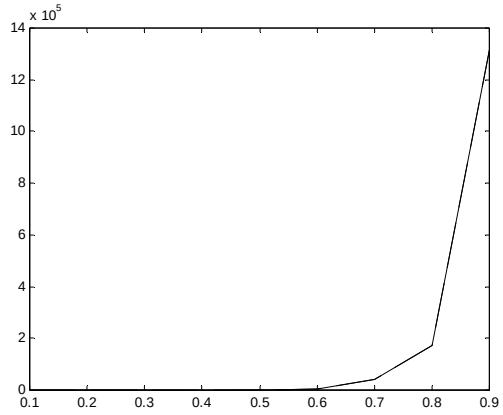
p=0.3



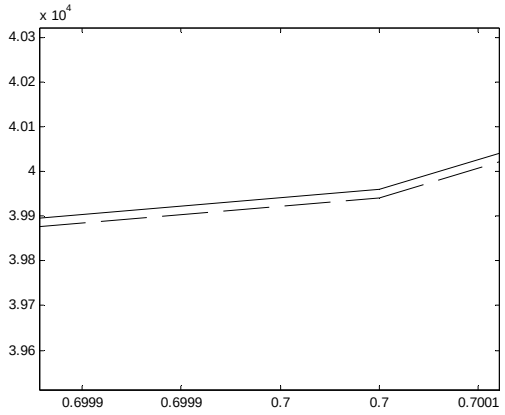
p=0.3 (zoom)



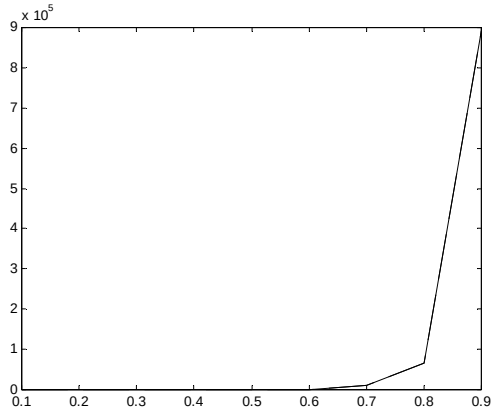
p=0.4



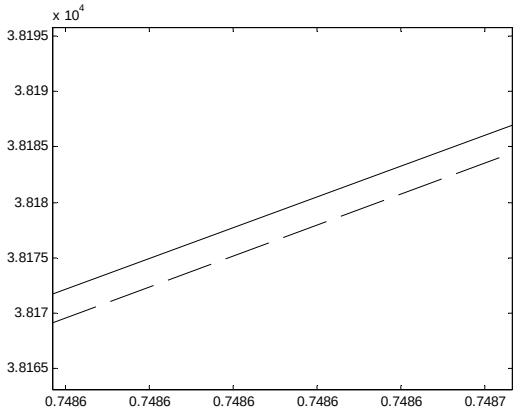
p=0.4 (zoom)



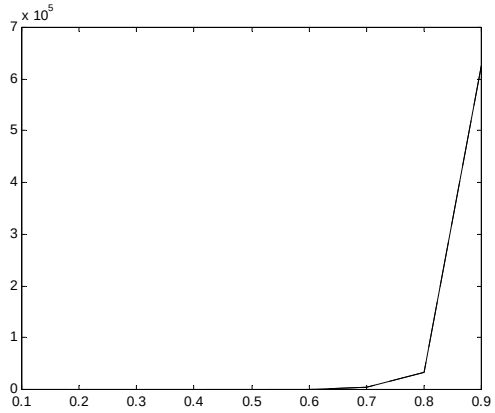
p=0.5



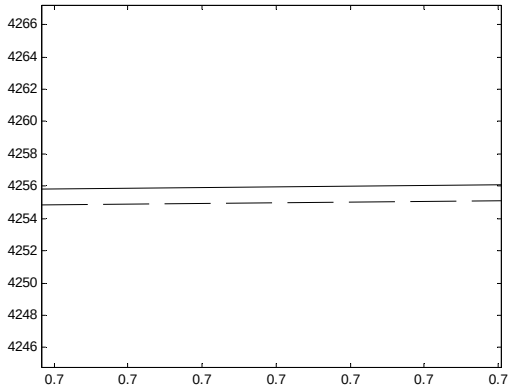
p=0.5 (zoom)



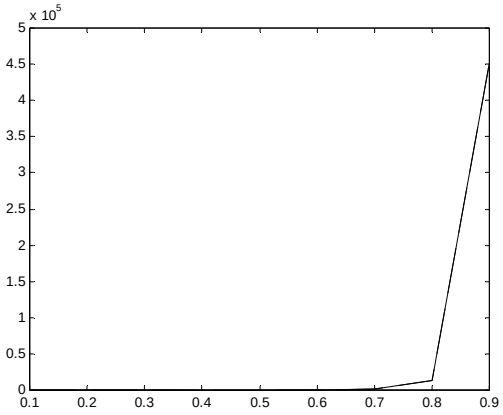
p=0.6



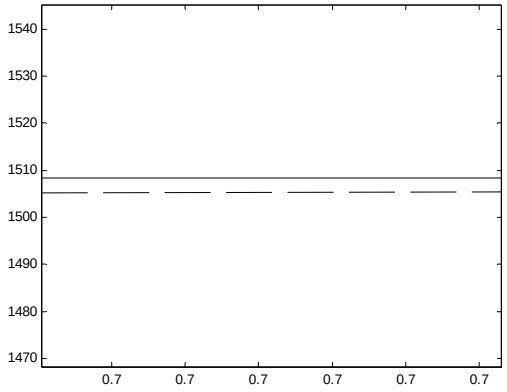
p=0.6 (zoom)



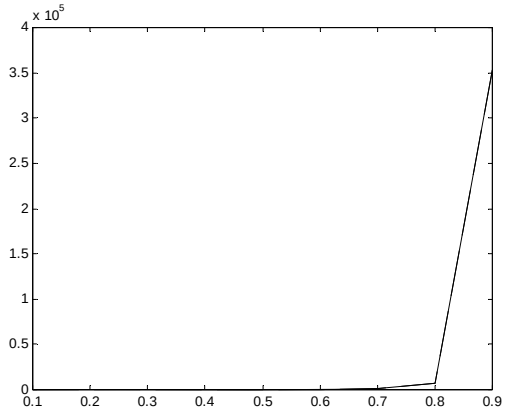
p=0.7



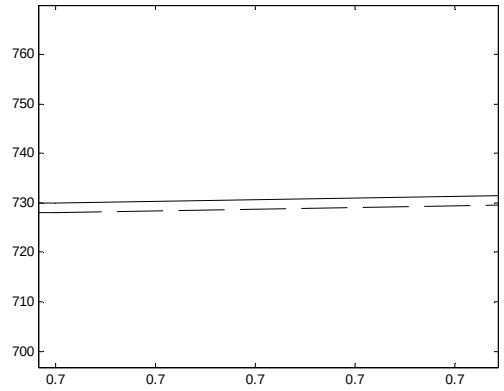
p=0.7 (zoom)



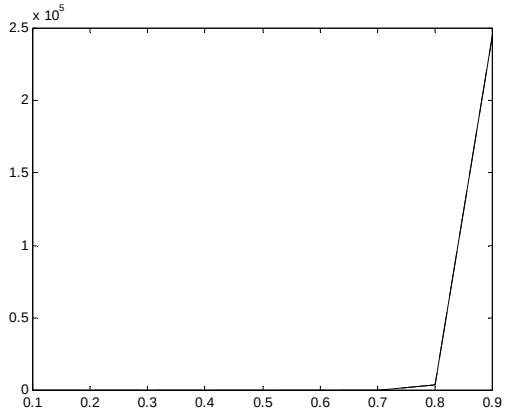
p=0.8



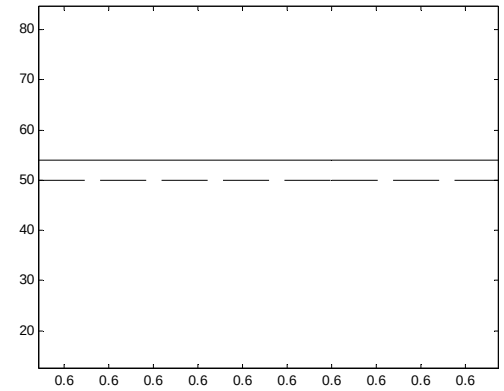
p=0.8 (zoom)



p=0.9



p=0.9 (zoom)

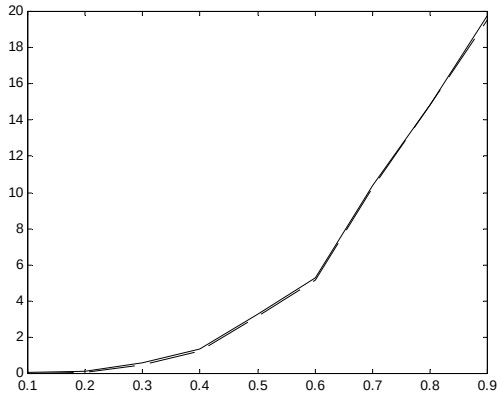


5.3 Runtime plots

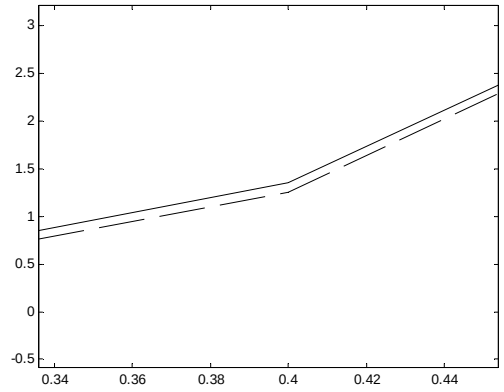
In the diagrams presented in this section, we have on x-axis the percentage of allowed tuples per constraint q and on the y-axis the CPU time in seconds needed for the execution of the FC algorithm in any case.

We also present 9 plots each one for every value of p from 0.1 to 0.9. With the continuous line we represent the improved FC algorithm. With the dashed line we represent the AC plus FC algorithm. Since the curves in any plot are very close, we present also a zoom plot.

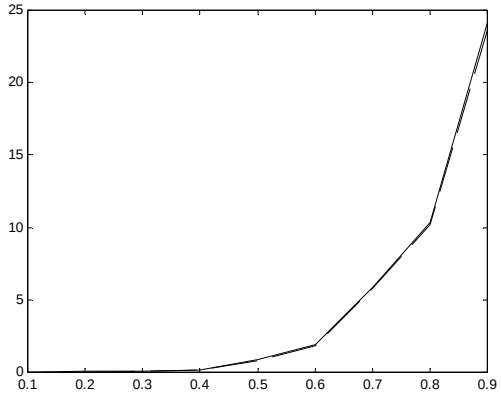
p=0.1



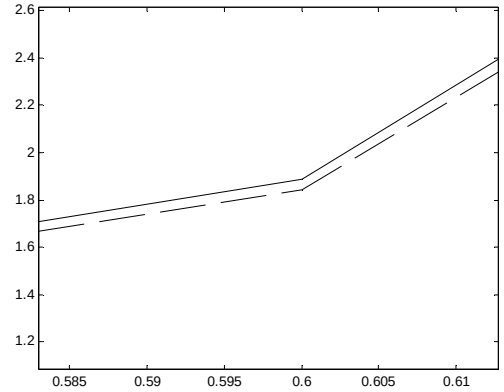
p=0.1 (zoom)



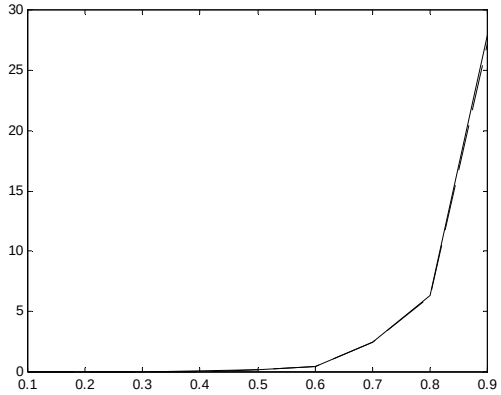
p=0.2



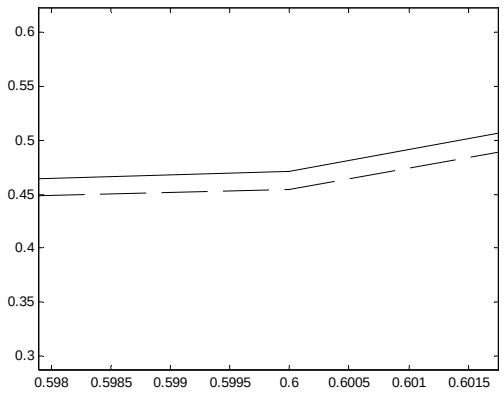
p=0.2 (zoom)



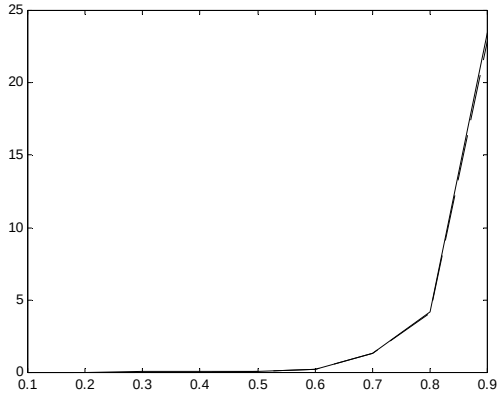
p=0.3



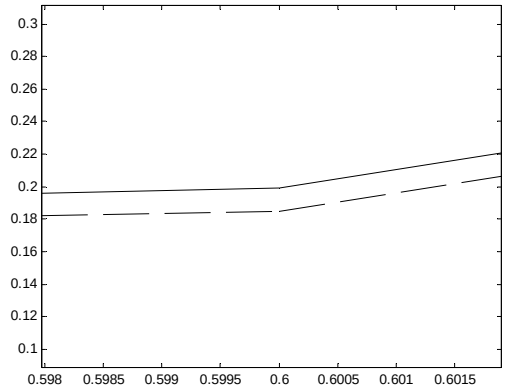
p=0.3 (zoom)



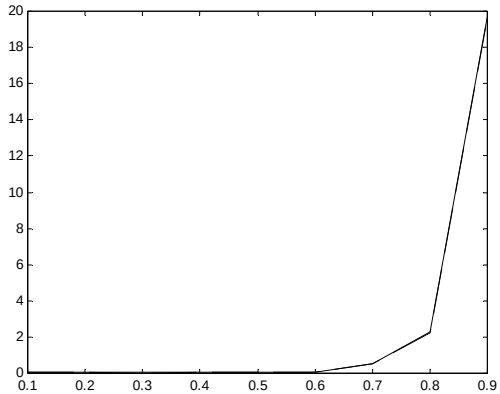
p=0.4



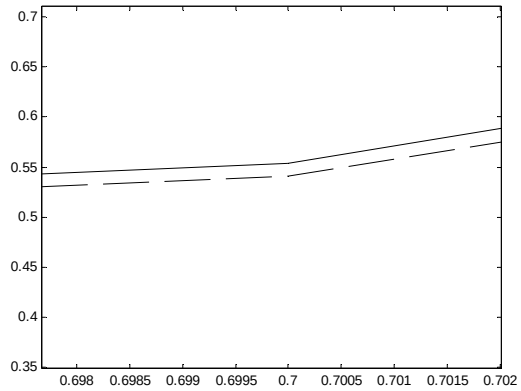
p=0.4 (zoom)



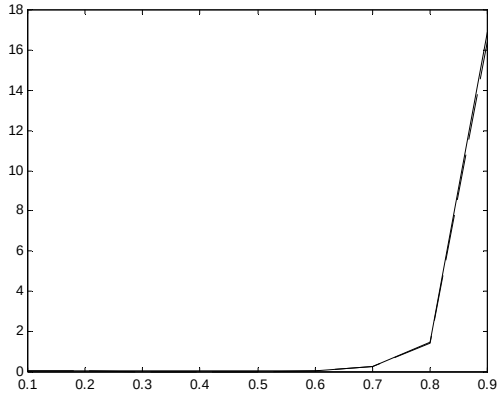
p=0.5



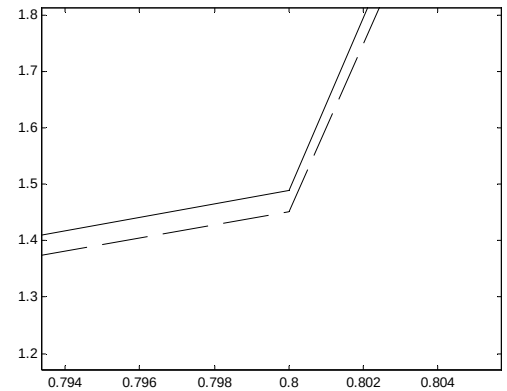
p=0.5 (zoom)



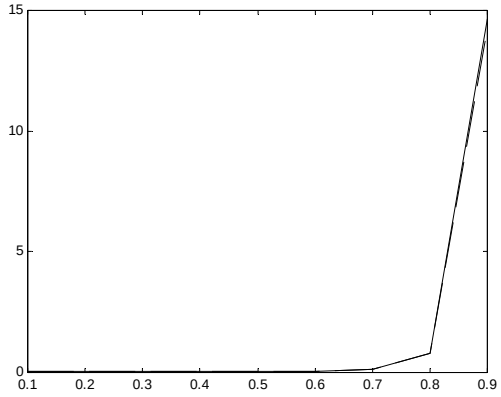
p=0.6



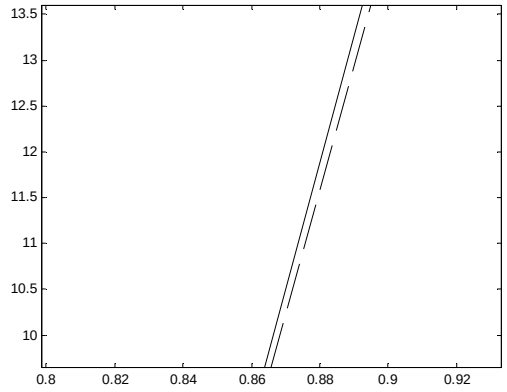
p=0.6 (zoom)



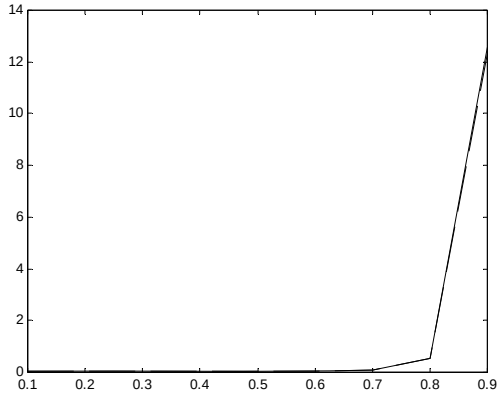
p=0.7



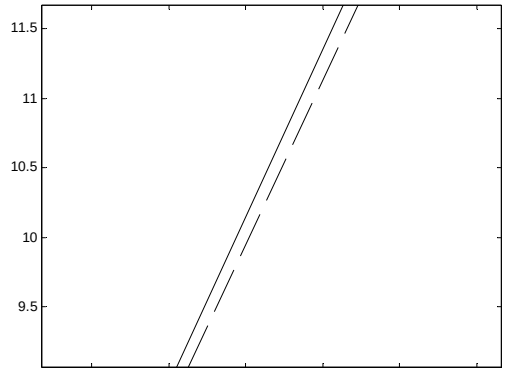
p=0.7 (zoom)



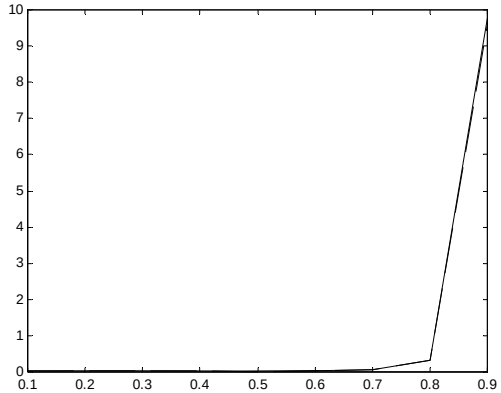
p=0.8



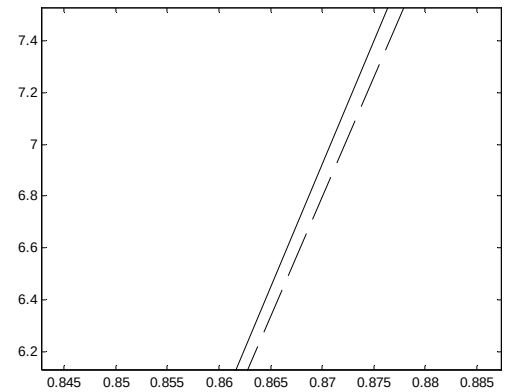
p=0.8 (zoom)



p=0.9



p=0.9 (zoom)



5.4 Experiments Evaluation

In many parameter settings of the above experiments, Walsh's FC algorithm has compute wrong value for the optimal satifiability. In some cases this error is over 50% of the instances.

As can be derived from the above plots, when the percentage q of the allowed tuples per constraint is small, then the problems are easy. As q increases, problems become much harder. This can be explained as follows. When q is small, which means that there are few allowed tuples, then the problems are over-constrained and cannot be satisfied. That is, there is no policy that satisfies all constraints. This is verified quickly by FC or even by AC preprocessing. As q increases, the number of policies that satisfy the constraints is also increased. Thus, to find the optimal satisfaction we need to search many more nodes in the search tree. As we can see from the plots, there is a point where we notice a sharp transition in the difficulty of the problems. This transition is sharper for denser problems (i.e. ones with high p). This is a typical phenomenon in many randomly generated combinatorial problems, known as a phase transition. Further experiments with larger problems are required to better understand the phase transition behavior of SCSPs.

In any case as we can see from the zoom plots, the use of Arc Consistency as a preprocessing step results in a small downfall of the search cost and runtime. We expect the benefits gained by the application of AC before or during search to be much higher as the size of the problems increases.

Chapter 6

Conclusions

In this thesis we studied the Stochastic Constraint Satisfaction Problem, an extension of the Constraint Satisfaction Problem which includes both decision variables (which we can set) and stochastic variables (which follow some probability distribution). The framework is designed to take advantage of the best features of traditional constraint programming, stochastic integer programming, and stochastic satisfiability. It can be used to model a wide variety of decision problems involving uncertainty and probability.

Our work has followed the semantics for stochastic constraint programs based upon policies. These determine how decision variables are set depending on earlier decision and stochastic variables. We have studied the algorithms presented in [Walsh02] and have shown that the FC algorithm suffers from a flaw in the way forward checks are made which can result in erroneous answers. We have corrected the flaw and proposed an improved version of the FC algorithm that achieves stronger pruning.

Moreover, we have described GAC and MAC algorithms for Stochastic CSPs that exploit specialized pruning rules to refute branches of the search tree and thus save search effort. In this way our work generalizes the work of Walsh where only binary constraints were considered.

As future work we intend first to complete the work presented here by implementing and testing the MAC algorithm and also incorporating in MAC the extension for chance

constraints. Then we would like to explore ways to increase the efficiency of backtracking-based algorithms by defining and implementing features such as intelligent backjumping, pure literal detection, and higher consistency enforcement. Also, the implementation of efficient propagation procedures for specialized constraints should be considered. Finally, we would like to explore alternative ways to solve stochastic CSPs by borrowing ideas from the fields of stochastic programming, on-line optimization, and local search.

References

[Bess05] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, Yuanlin Zhang: An optimal coarse-grained arc consistency algorithm. Artif. Intell. 165(2): 165-185 (2005)

[Dan94] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In Proceedings of ECAI-94, pages 125{129, Amsterdam, The Netherlands, 1994.

[Dech92] Dechter, R.: 1992, 'Constraint Networks'. In: S. Shapiro (ed.): Encyclopedia of Artificial Intelligence. John Wiley & Sons, pp. 276-285.

[Falt02] Faltings, B. and S. Macho-Gonzalez: 2002, 'Open Constraint Satisfaction'. In: Proc. of the 8th International Conference and Principles of Constraint Programming (CP-02). Ithaca, New York, USA, pp. 356-370.

[Farg93] Fargier, H. and J. Lang: 1993, 'Uncertainty in Constraint Satisfaction Problems: A Probabilistic Approach'. In: Proc. of the European Conference on Symbolic and Quantitative Approaches of Reasoning under Uncertainty (ECSQARU-93). Grenade, Spain, pp. 97-104.

[Farg96] Farfier, H., J. Lang, and T. Schiex: 1996, 'Mixed Constraint Satisfaction: a Framework for Decision Problems under Incomplete Knowledge'. In: Proc. of the 13th National Conference on Artificial Intelligence (AAAI-96). Portland, OR, USA, pp. 175-180.

[Fowl00] Fowler, D. and K. Brown: 2000, 'Branching Constraint Satisfaction Problems for Solutions Robust under Likely Changes'. In: Proc. of the 6th International Conference on Principles and Practice of Constraint Programming (CP-00). Singapore.

[Littm01] Littman, M., S. Majercik, and T. Pitassi: 2001, 'Stochastic Boolean Satisfiability'. Journal of Automated Reasoning 27(3), 251-296.

[Mac77a] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mack92] Mackworth, A.: 1992, 'Constraint Satisfaction'. In: S. Shapiro (ed.): Encyclopedia of Artificial Intelligence. John Wiley & Sons, pp. 285-293.

[McG79] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphism. *Information Science*, 19:229–250, 1979.

[Mitt90] Mittal, S. and B. Falkanhainer: 1990, 'Dynamic Constraint Satisfaction Problems'. In: Proc. of the 8th National Conference on Artificial Intelligence (AAAI-90). Boston, MA, USA, pp. 25-32.

[Sabin98] Sabin D. and E. Freuder: 1998, 'Detecting and Resolving Inconsistency and Redundancy in Conditional Satisfaction Problems'. In: Proc. of the CP-98 Workshop on "Constraint Problem Reformulation". Pisa, Italia.

[Walsh02] Walsh, T.: 2002, 'Stochastic Constraint Programming'. In: Proc. Of the 15th European Conference on Artificial Intelligence (ECAI-02). Lyon, France.