A FRAMEWORK FOR EXPERIMENTS IN CSPs

by

George Katsirelos

A thesis submitted in conformity with the requirements
for the degree of MSc
Graduate Department of Computer Science
University of Toronto

Abstract

A framework for experiments in CSPs

George Katsirelos

MSc

Graduate Department of Computer Science

University of Toronto

2001

The Constraint Satisfaction Problem (CSP) is an NP-complete problem, which allows flexible and intuitive representation of real-world problems.

In this thesis, we present a consistent view of existing search algorithms and variable ordering heuristics. We then examine several aspects of building a CSP solver to accommodate their requirements in a generic way. We demonstrate the feasibility of using this solver to perform experiments in common problems. Finally, we present and examine the performance of a new heuristic method to manipulate existing CSP models by conjoining constraints in order to improve their performance when used with the GAC search algorithm.

Contents

Chapter 1

Introduction

1.1 The constraint satisfaction problem

A Constraint Satisfaction Problem P is a tuple (V, D, C), where

- V is a set of variables
- D is a set of domains indexed by variable
- C is a set of constraints, where each constraint is over a set of variables, denoted $VarsOf(C)$.

We can *assign* to a variable V a value x from its domain, which we denote $V \leftarrow x$. An *assignment* $\mathcal{A}$ is a set $\{V_1 \leftarrow x_1, \ldots, V_m \leftarrow x_m\}$, such that $V_i \neq V_j, i \neq j$. This means that no variable can be assigned more than one value. We define $VarsOf(\mathcal{A})$ to be $\{V_1, \ldots, V_m\}$.

A constraint C is a set of assignments to $VarsOf(C)$. Each tuple that belongs to the constraint is said to *satisfy* it. A constraint is *fully instantiated* by an assignment $\mathcal{A}$ if $VarsOf(C) \subseteq VarsOf(\mathcal{A})$. Moreover $\mathcal{A}$ satisfies C if the subset $\mathcal{A}'$ of $\mathcal{A}$ that has $VarsOf(C) = VarsOf(\mathcal{A})$, satisfies C.

The *arity* of a constraint is $\| VarsOf(C) \|$. The arity of a problem is the maximum arity of its constraints. Therefore, a binary CSP is one that only has binary or unary constraints.

Often binary CSPs are represented by a *constraint graph*. In it, vertices represent variables of the CSP and edges represent constraints between variables. This representation has no information about the structure of the individual constraints, but it lends itself to an analysis of the problem using a graph theoretic approach. Non-binary CSPs can be represented using a hypergraph.

A *solution* to a CSP is an assignment of values to all the variables, such that it satisfies all constraints.

The satisfiability problem is a special case of the constraint satisfaction problem, where the domains of all the variables are $\{0, 1\}$, even though there are other ways to encode a SAT problem as a CSP [34, 2]. A polynomial translation can be performed from CSP to SAT, as well [34, 15]. Therefore, the CSP is an NP complete problem. However, there are instances of the problem that are solvable and useful. In fact, CSPs are widely applicable, as we will see in the next section.

1.2 Applications

1.2.1 Optimal Golomb Ruler

A golomb ruler is a set of n numbers, called marks, such that the $n(n-1)/2$ differences between every pair of them is unique. An optimal golomb ruler is one such that $max_m - min_m$ is minimal, where max_m and min_m are the maximum and minimum marks, respectively.

It has been noted that finding an optimal golomb ruler is very hard [31], even for small numbers of marks. However, it has applications in many areas of engineering, such as wireless communications.

This problem will be discussed in more detail later, when we will examine how it can best be represented and solved as a CSP.

1.2.2 Assembly line sequencing

The problem of assembly line sequencing [3] involves finding a sequence that items will be placed on an assembly lines, while not exceeding the capacity of various resources in the line and at the same time completing the sequence in the smallest time possible.

This problem obviously has wide applications in manufacturing environments. In addition, it is possible to come up with numerous variations on the basic problem, such as:

- Variety for the workers on the line, to maintain their interest and skill sets.
- Limitations can be set on the possible transitions between items. For example, for a painting machine, an item that must be painted white should not be placed immediately after one that should be painted red, so that it does not end up with a pink color.

1.2.3 Scheduling

The car sequencing problem is a special case of the scheduling problem [10], which involves assigning jobs to machines. There are many constraints that limit the possible solutions to this problem, including what kinds of jobs each machine can handle and in what succession, what jobs need to be completed and so on.

In addition, scheduling is a problem that can have a vast number of solutions, all acceptable given the constraints. However, only some of them may be desirable.

Therefore, scheduling is suitable for the use of techniques in optimization, as well as *hard* and *soft* constraints (hard constraints need to be satisfied for a solution to be

reported, whereas soft constraints, when violated, increase a penalty value for the solution and the goal is to minimize the penalty [29].)

1.3 Research in CSPs

Research in the area of CSPs is very experimentally intensive. This is because there is usually no way to completely determine how a new algorithm or dynamic variable ordering heuristic will work without actually trying it in real world problems. Moreover, the performance characteristics of each algorithm can vary widely between different problems. The best way to solve a given problem generally involves choosing a combination of algorithm, heuristic and representation. The best representation for one algorithm may not be the best for another.

1.3.1 Algorithms and Heuristics

As will be discussed in chapter 2, there are many different approaches to solving CSPs.

The main aspects that affect the efficiency of solving a CSP, given a specific representation, are

- The choice of algorithm
- The heuristic for choosing which assignment to make next.

Even though theoretical comparisons of algorithms and heuristics are possible, the actual performance of an algorithm is a function of the tradeoffs that it makes and how effective they are for a given problem. In chapter 2, we will discuss these tradeoffs and examine how they affect performance.

1.3.2 Representations

By far the most challenging aspect of CSPs is coming up with an efficient representation to solve a CSP. These are the most important features of a representation:

- Number of variables
- Size of domains
- Uniformity of domain sizes
- Number of constraints
- Arity of constraints
- Distribution of constraints in the constraint graph
- Pruning strength of constraints

All the above characteristics influence the effectiveness of a representation. Moreover, we have to balance other characteristics, depending on the algorithm used. For example, intelligent backtracking algorithms can deduce better backjumps with certain type of constraints. Also, we may be able to do more efficient constraint propagation by taking advantage of domain knowledge. Finally, adding redundant constraints, although it increases the cost of consistency checks, can prove beneficial by enabling the constraint propagation algorithms to perform more pruning.

Consider the example of the n-queens problem [23]. One possible representation is to represent each square with a variable whose domain is $\{0, 1\}$, which means that there is or there is not a queen in that square, respectively. In addition, there will be a constraint between every pair of variables, that only allows a queen to be placed in both squares if they do not "see" each other. So, for the 4-queens problem, there will be 16 variables and 120 binary constraints. In other words, this representation needs $O(n^2)$ variables and $O(n^4)$ constraints.

On the other hand, we can represent the problem using one variable for each column of the chess board. The domain of each variable will be $\{1 \ldots n\}$, representing the row where a queen will be placed on that column. We also post a constraint between every two variables, which will be true if the placement of the queens in the corresponding columns is such that they do not "see" each other. In other words, we use $O(n)$ variables and $O(n^2)$ constraints. This representation uses much fewer variables and constraints, by taking advantage of the knowledge that there can be only one queen at each column of the board.

One common benchmark problem, that is much more difficult and realistic that the n-queens problem discussed above, is the Optimal Golomb Ruler problem. Smith et al. [31] study various representations and find that a good representation is one that has $O(m^2)$ variables, instead of $O(m)$, but uses a more efficient constraint propagation algorithm. It also uses ternary constraints and one constraint of very large arity, whereas the original representation uses quaternary constraints. A theoretical comparison showed that either representation used could, in different situations, be better than the other. Their empirical results, however, show that the representation with ternary constraints was consistently much better. Moreover, this paper shows that, contrary to common practice, the Brélaz variable ordering heuristic actually performs worse than a static lexicographic heuristic for this problem. We will discuss these results again in chapter 4.

1.4 Contributions

It becomes clear that research in the field of CSPs is heavily dependent on access to an experimental tool. However, there is a wide array of available algorithms, heuristics, as well as transformations that can be done on a model, so that it is not obvious how to implement them all in a consistent and extensible way.

In this thesis, we present a consistent view of existing algorithms and heuristics in

chapter 2, examine several aspects of building a CSP solver to accommodate these and address them all in a generic way in chapter 3. We then demonstrate the feasibility of using this solver to solve some common benchmark problems, in chapter 4.

The purpose of this work is to provide a framework for supporting the empirical investigations required to do research in this area.

Chapter 2

Previous Work

2.1 Backtracking

One way to generate all solutions for a CSP is to generate all the possible sets of assignments and discard those that fail to satisfy all the constraints.

The idea behind backtracking [8] is that instead of generating an assignment to the variables and then checking it, we can check it as we create it. A backtracking algorithm assigns a value to a variable, then tries to assign another variable and so on. With each assignment that it makes, it makes sure that all the fully instantiated constraints are satisfied. It continues until there are no more variables left, in which case it has found a solution, or a constraint that has been falsified. In the first case it reports the solution found and continues as if it had failed. In the second case, it tries another value for the current variable. When all values in the variable's domain are exhausted, it tries the next value for the previous variable. The search is over when all values of the first variable have been exhausted.

Checking sets of assignments as we build them can save an exponential amount of time in solving the problem. Consider the case of a problem with n variables, all of which have the same domain size d. Consider, further, that the backtracking algorithm instantiates

k variables when it finds that this partial assignment falsifies some constraint. It will now backtrack, trying other values for the k variables that it has instantiated. The number of assignments that it has avoided considering is $(n-k)^d$.

Backtracking can be seen as a search in a tree, where each node is a partial assignment and its children are nodes whose corresponding assignment is the same except they have one more assigned variable. The leafs of the tree are complete assignments or assignments that fail to falsify some constraint.

Seen this way, the benefit of backtracking over generate-and-test is that it reduces the size of the explored search tree by avoiding searching some subtrees that contain no solution. These subtrees can be of exponential size.

2.2 Constraint Propagation

2.2.1 Local consistency

Using any of the algorithms that we will discuss later, the time that it takes to find a solution for a CSP depends on the size of the search tree, which is, in the worst case, the size of the product of the domains of all the variables in the problem. We can reduce the size of the search tree by examining the constraints individually and pruning values that are locally inconsistent.

Local inconsistency is the inverse of local consistency. We will consider the following three types of local consistency.

- Node consistency

  A problem is node consistent when all the values in the domains of its variables satisfy all unary constraints.

- Arc consistency [19]

A value is arc consistent for a binary constraint when there is a satisfying set of assignments to the variables of the constraint that contains this value. A constraint is said to be arc-consistent when all of the values of its variables are arc-consistent. A problem is arc-consistent when all its binary constraints are arc-consistent.

- Generalized arc consistency [20]

  A value is arc consistent for a constraint when there is a satisfying set of assignments to the variables of the constraint that contains this value. A constraint is said to be arc-consistent when all of the values of its variables are arc-consistent. A problem is arc-consistent when all its constraints are arc-consistent.

When enforcing these forms of local consistency, we find inconsistent values and prune them.

Node consistency is easier to implement, as we only have to consider unary constraints. Therefore, finding node inconsistent values has time complexity only $O(d)$ for the domain of one variable, where d is the size of the variable's domain. Since arc consistency (AC) deals with binary constraints, it has to examine d^2 pairs of values, therefore its complexity is $O(d^2)$. Similarly, generalized arc consistency (GAC) has complexity $O(d^k)$, where k is the arity of the constraint being checked.

On the other hand, since node consistency only considers unary constraints, it can generate little, if any, pruning. AC considers binary constraints. while GAC considers all constraints. Each is more powerful and can generate more prunings.

Another thing to note is that AC and GAC have cascading effects. This means that when a value is pruned from the domain of one variable, all the constraints in which this variable participates must be reexamined for AC (or GAC). Consider three variables V_1, V_2, V_3 with domains $\{1,2,3\}$ and constraints $C_{1,2} = \{\{1,2\},\{2,3\}\}$ and $C_{2,3} = \{\{2,3\}\}$. Enforcing AC on $C_{1,2}$ will prune $V_1 \leftarrow 3$ and $V_2 \leftarrow 1$. Similarly making $C_{2,3}$ arc consistent will prune $V_2 \leftarrow 1, V_2 \leftarrow 3$ and $V_3 \leftarrow 1, V_3 \leftarrow 2$. If, however, we

enforce AC on both of them, pruning $V_2 \leftarrow 3$ will force reexamination of $C_{1,2}$ and that will cause $V_1 \leftarrow 2$ to be pruned as well.

What we see is a trade-off between processing time for enforcing different types of local consistency and how much pruning (therefore reduction in the size of the search tree) they achieve.

We can further reduce the size of the search subtree below the current node by enforcing some form of local consistency at each node [14, 28].

When we make an assignment, we effectively reduce the arity of all constraints that this variable participates in by 1. The new constraints are satisfied by the subset of tuples from the original constraints that contain the value we just assigned. This means that each assignment can potentially generate new binary and unary constraints, so that we can apply NC and AC on the subproblem.

The above techniques allow us to view enforcing local consistency as a dynamic procedure rather than a static one that only occurs before we start solving the problem, therefore enabling us to reduce the size of the search subtree further at each node that we visit.

Similarly, when we make an assignment, it can be viewed as reducing its current domain to a singleton domain. This means that the rest of its values get pruned. Therefore, we now have a set of prunings that GAC can propagate.

Note, however, that when a constraint propagation algorithm prunes a variable, it has done so because it has discovered a reason for its pruning. Therefore, when this reason no longer exists, these values should be restored to the domain of its variable. That reason can be complex, but invariably contains the assignment that was made when that value was pruned. Therefore, we introduce the notion of the *prune level* of a value. When the assignment made at level l causes a value to be pruned, its prune level is set to l. When the algorithm backtracks and undoes the assignment made at l, all values that were *pruned to level* l, are restored to their original domains.

On the other hand, constraint propagation at each node can mean that we end up doing more work than is necessary. This can happen when a constraint propagation algorithm prunes the domain of a variable that is never assigned in the subtree below. In this case, all the checks that were made to prune the domain of this variable were wasted.

2.2.2 Forward Checking (FC)

Forward checking [17] is an algorithm that tries to enforce node consistency at each node that it visits.

After it makes an assignment $V_i \leftarrow x$, it goes over all the constraints C in which V_i participates that only have one uninstantiated variable left (after the assignment $V_i \leftarrow x$). For every value y of the unassigned variable V_j, it takes the current assignment $\mathcal{A}$ and checks the constraint against the assignment $\mathcal{A} \cup V_j \leftarrow y$. If the check fails, the value y is pruned from the domain of V_j.

2.2.3 Maintaining Generalized Arc Consistency(MGAC)

MGAC [19] does exactly what its name implies: it enforces generalized arc consistency at each node visited by the search procedure.

To enforce GAC on a constraint, the algorithm goes over all the variables participating in this constraint. For every value of each variable, it checks to see that there is a tuple that contains this value and satisfies the constraint. This is called a *supporting* tuple for the value, or simply *support*. If there is no supporting tuple for a value, it gets pruned from the domain of its variable.

To keep the problem GA consistent, the algorithm has to go over all the constraints at least once. When it prunes a value, however, it can mean that another value has lost its supporting tuple. So, it has to make sure again that all the values that may have had support on it still have some support, otherwise they must be pruned.

A naive way to do that would be to go through all the constraints and ensure that each value in every domain has a supporting tuple and repeat the process until no more values get pruned. This method is called AC-0.

A significantly better way to do that is to maintain a queue of pairs $< C, V >$. Initially, all such pairs are inserted into the queue. The algorithm repeatedly removes pairs from the queue and checks the GA consistency of the values of V with respect to C. Whenever a value gets pruned from the domain V, all $< C', V' >$ pairs such that $C' \neq C$, $V' \in VarsOf(C')$ and $V' \neq V$ are inserted into the queue if they are not already there. This way, if a value x of variable V gets pruned and it is possibly part of a supporting tuple for value x' of variable V', all values of V' are going to be reexamined for all constraints that constrain both V and V'. This method, called AC-3 [19] enforces GAC much faster than AC-0.

It should be noted that other structures can be used instead of the queue, such as a stack or a priority queue. That is, the algorithm does not depend on the FIFO property of the queue.

There are even better ways to keep track of supporting tuples and further improve AC-3. AC-4 [22] has been proposed and has been proven to be optimal in time complexity. Variations up to AC-7 have also been proposed (AC-5 [18], AC-6 [4], AC-7 [5], lazy AC-7 [30]) but they only improve the running time by a constant factor and have a significant space complexity.

2.2.4 Maintaining Arc Consistency (MAC)

MAC [19] is just a special case of MGAC, where only constraints that have two uninstantiated variables left are inserted into the queue.

2.3 Intelligent Backtracking

Whereas constraint propagation tries to reduce the size of the search tree, intelligent backtracking tries to reduce the size of the *explored* search tree. That is, it tries to reduce the number of nodes that it actually visits, even though it does not necessarily prune any values.

An intelligent backtracking algorithm tries to discover a reason why the current subtree fails to contain a solution. The search can then backtrack far enough to invalidate this reason. To understand how this can be accomplished, we introduce the concept of *nogoods*.

A nogood is a set of assignments that is not part of any solution. Note that any superset of a nogood is also a nogood.

There are an exponential number of nogoods that an algorithm can learn. The algorithms that we discuss associate each assignment with the level of the search tree where it was made. The nogoods that they learn are then represented as sets of levels. This representation restricts the set of nogoods that are learned to subsets of the current assignment.

In fact, these algorithms learn two kinds of nogoods:

- A nogood that directly violates a constraint.

  These are encountered whenever the search finds a value that violates a constraint. They consist of all the assignments made to variables that belong to the constraint that was violated.

- A nogood that results from the unioning of other nogoods.

  We will discuss the following two types of unioning nogoods

  - Unioning nogoods that cover the domain of a variable. When we have a set of nogoods that cover the domain of a variable V, we can union these nogoods

minus the assignments to V and get a new nogood.

In other words, when we exhaust the domain of V and discover a reason for each of its values to not be a part of any solution, the union of the nogoods discovered presents a reason why the rest of the assignments made so far cannot be extended to a solution. This is because a solution has to contain an assignment to every variable and we have found a reason why there exists no solution that extends the current assignment with an assignment to V.

- Constraint-filtered unioning. When we have a constraint $C_{V_1V_2}$ and a set of nogoods that cover the support (as defined for GAC) of $V_1 \leftarrow x$ on V_2, we can get a new good from the union of these nogoods minus the assignments to V_2 plus $V_1 \leftarrow x$. If, for example, the supports of $V_1 \leftarrow x$ on V_2 are $V_2 \leftarrow y_1$ and $V_2 \leftarrow y_2$, and we have discovered the nogoods

$$\{V_2 \leftarrow y_1, V_3 \leftarrow z\}$$
$$\{V_2 \leftarrow y_2, V_4 \leftarrow z'\}$$

then we learn the nogood

$$\{V_1 \leftarrow x, V_3 \leftarrow z, V_4 \leftarrow z'\}$$

Intuitively, this means that if we have discovered a reason why the current set of assignments cannot be extended to a solution that contains the set of supports of $V_1 \leftarrow x$ on V_2, then this is also a reason why the current set of assignments cannot be extended to a solution that contains $V_1 \leftarrow x$. This can be explained by noting that any solution that contains $V_1 \leftarrow x$ also has to contain its supporting values.

A *conflict set* for a value x of a variable V is a set of assignments $\mathcal{A}$, such that $\mathcal{A} \cup \{V \leftarrow x\}$ is a nogood.

We can also add pruning to an algorithm that maintains conflict sets. Whenever we discover a conflict set for $V \leftarrow x$, we can prune that value to the deepest level where an assignment of the conflict set was made. In other words, when the search discovers a conflict set CF for $V \leftarrow x$, it means that $CF \cup \{V \leftarrow x\}$ is a nogood. As long as all the reasons for this being a nogood are valid, the algorithm does not need to try this assignment, since it will fail. Therefore, x can be pruned from the domain of V and remain pruned until at least one of the assignments in CF is undone. This will occur at level $max(levelof(V')), V' \in CF$; and this will be the prune level of x.

2.3.1 Backjumping(BJ)

BJ [14] maintains one conflict set for each variable.

When it attempts to assign a value to a variable, it finds the earliest level l that this assignment becomes inconsistent. The conflict set that it discovers at this point consists of all the assignments at level l and above in the tree. Since the structure of all the conflict sets is always the same (all the levels from 1 to l), it simply stores l.

As an example, consider that BJ tries to make the assignment $V \leftarrow x$, but this assignment fully instantiates and violates constraints $C_{V_1,V_3,V}$ and $C_{V_2,V_4,V}$, while variables V_1, V_2, V_3, V_4 are instantiated at level 1, 2, 3 and 4 respectively. In this case, the earliest point where the assignment to V is inconsistent with the prior assignments is 3, therefore the conflict set BJ discovers for $V \leftarrow x$ consists of the assignment at levels $\{1, 2, 3\}$.

The conflict set for a variable is the maximum level l discovered for all of its values. BJ can jump back to l when it exhausts the domain of that variable.

Suppose that, in the above example, BJ tries all values in the domain of V and finds that l is the level it should jump back to. By doing this, it has discovered a new set of nogoods of the form:

$$\{V_1 \leftarrow x_1, \ldots, V_l \leftarrow x_l, V \leftarrow y_1\}$$
$$\vdots$$
$$\{V_1 \leftarrow x_1, \ldots, V_l \leftarrow x_l, V \leftarrow y_d\}$$

where V_i is the variable that has been assigned at level i.

By unioning these nogoods, since they cover the domain of V, it gets a new nogood

$$\{V_1 \leftarrow x_1, \ldots, V_l \leftarrow x_l\}$$

Therefore, the new conflict set for $V_l \leftarrow x_l$ is the one that consists of all the assignments made at levels $1 \ldots l-1$.

This means that BJ is only able to jump back only once. To jump back further, it first has to explore another subtree. In other words, it can only jump back from leafs, not from internal nodes of the tree.

2.3.2 Conflict directed Backjumping(CBJ)

CBJ [24] improves on BJ by learning better nogoods. Instead of learning a nogood that only consists of all the assignments made at levels $1 \ldots l$, it takes advantage of the specific information provided by a failed constraint. Specifically, when the assignment $V \leftarrow x$ fails to satisfy a constraint $C_{V_{k_1},\ldots,V_{k_m},V}$, the conflict set that CBJ learns for x consists of the assignments made to the variables $V_{k_1}, \ldots, V_{k_m}$. This conflict set is unioned into the conflict set for the entire variable.

After iterating over the entire domain of the variable, the conflict set discovered for the variable is the union of the conflict sets for all its values. In other words, it is the

union of a set of nogoods that cover its domain, minus the assignments to this variable. Therefore, this conflict set is itself a new nogood.

CBJ can now jump back to the deepest level where an assignment in the conflict set of the variable was made. It is safe to do so, because as long as none of the assignments in the nogood are undone, there is no solution.

The nogood discovered minus the assignment at the level CBJ jumps back to is a conflict set for the assignment made at that level. Therefore, this conflict set can be unioned into the conflict set for the variable at the jumpback level.

Because CBJ maintains finer-grained conflict sets than BJ, it is possible for it to jump back not only from leafs, but also from internal nodes.

For example, suppose that variables V_1, V_2, V_3, V_5 have been instantiated at levels 1, 2, 3 and 5, respectively. Further, the algorithm has reached level 10, where it instantiates variable V with domain $\{a, b\}$. The assignment $V \leftarrow a$ violates the constraint $C_{V_1,V_2,V_5,V}$, while the assignment $V \leftarrow b$ violates the constraint $C_{V_2,V_3,V_5,V}$. The conflict set for $V \leftarrow a$ will be $\{1, 2, 5\}$ and the one for $V \leftarrow b$ will be $\{2, 3, 5\}$. The conflict set for V is the union of these, which is $\{1, 2, 3, 5\}$ and therefore CBJ jumps back to level 5 and sets the tentative conflict set for V_5 to $\{1, 2, 3\}$. Now, the algorithm assigns other values to V_5. Suppose that the other nogoods that it finds for V_5 are subsets of its current conflict set, therefore the conflict set after CBJ exhausts V_5's domain is still $\{1, 2, 3\}$. Now it will jump back to level 3. In contrast, BJ would also jump back to level 5, but it would then only be able to step back to level 4. CBJ is able to jump further back because it maintains more detailed conflict sets than BJ.

2.3.3 Value specific Conflict directed Backjumping (vsCBJ)

vsCBJ [1] improves upon CBJ by maintaining a conflict set for each value of a variable.

When it discovers a conflict set for a value, it simply keeps it for this value and does not union it into a single conflict set for the variable.

This allows it to take advantage of constraint-filtered unioning of nogoods to discover more powerful nogoods on backtrack. It produces the constraint-filtered nogood for the value assigned to the variable at the jumpback level, using either a constraint between the current variable and the jumpback variable or the universal constraint. It can also use a constraint which has all of its variables, except the two in question, assigned at levels above the jumpback level.

As an example, consider the case of a CSP, where the assignment $V \leftarrow x$ is made at level l_1, $V' \leftarrow x'$ is made at level $l_2 > l_1$ and at level $l_3 > l_2$, the domain $\{a, b, c\}$ of variable V'' is exhausted, without finding a solution in the subtrees. Moreover, the solver determines that it is safe to jump back to level l_2. Using CBJ, it will now set the conflict set of $V' \leftarrow x'$ to be the union of the conflict sets for $V \leftarrow a, V \leftarrow b, V \leftarrow c$. Consider however, if there is a constraint $C_{V,V',V''}$ and the tuple $\{V \leftarrow x, V' \leftarrow x', V'' \leftarrow c\}$ does not satisfy it. This means that as long as there exists a reason for $V'' \leftarrow a$ and $V'' \leftarrow b$ to be invalid and the assignment $V \leftarrow x$ is not reverted, the assignment $V' \leftarrow x'$ will also be invalid, regardless of the status of $V'' \leftarrow c$. In other words, the conflict set for $V' \leftarrow x'$ can be set to be union of the nogoods of the values of variable V'', filtered by the constraint $C_{V,V',V''}$:

$$CF_{V' \leftarrow x'} = V \leftarrow x \cup CF_{V'' \leftarrow a} \cup CF_{V'' \leftarrow b}$$

2.4 Combining Constraint Propagation with Intelligent Backtracking

Constraint propagation and intelligent backtracking utilize two different techniques to speed up the search. Constraint propagation tries to reduce the size of the search tree below the current node, while intelligent backtracking tries to reduce the size of the explored tree.

We can combine the two approaches to achieve even greater speed ups. Intelligent backtracking algorithms discover nogoods for values in two points in the search:

- When checking that the current assignment is consistent
- When backtracking

Constraint propagation discovers nogoods only when enforcing some form of local consistency after making an assignment. The nogood that it discovers for the values that it prunes consists only of the level at which they were pruned. In other words, it is the same type of nogood that BJ discovers: one that consists of all the assignments made until the current level. We can apply the same reasoning as we did for BJ and improve FC and GAC first to FCCBJ and GACCBJ and then to CFFC and CFGAC.

2.4.1 FCCBJ, GACCBJ

FCCBJ [24] and GACCBJ [25] perform the same kind of pruning as FC and GAC. The only difference is that when they prune a value from the domain of a variable, they use the constraint that caused the pruning to generate a conflict set for that value and union it into the conflict set of the corresponding variable.

FCCBJ can simply set the conflict set of the pruned value to be the set of assignments made to the rest of the variables of the constraint that caused the pruning. When doing GACCBJ, however, it is not as simple to know exactly what caused the pruning. Therefore, the conflict set for the pruned value is set to be the set of assignments made to variables of the constraint being checked unioned with the conflict sets of the rest of the pruned values in the variables of the constraint.

To demonstrate this, consider enforcing GAC on the constraint C_{V_1,V_2,V_3}, with all three variables having the original domain $\{1,2,3\}$, after having pruned $V_3 \leftarrow b$ and $V_3 \leftarrow c$ and making the assignment $V_1 \leftarrow b$. The constraint has the following satisfying tuples:

$$\{ \quad \{V_1 \leftarrow a, V_2 \leftarrow a, V_3 \leftarrow a\}$$

$$\{V_1 \leftarrow b, V_2 \leftarrow a, V_3 \leftarrow c\}$$

$$\{V_1 \leftarrow b, V_2 \leftarrow a, V_3 \leftarrow b\} \quad \}$$

At this point, $V_2 \leftarrow a$ should be pruned. Its conflict set has to include, besides the conflict set derived from the previously assigned variables of the constraint, the conflict set for $V_1 \leftarrow a$, $V_3 \leftarrow b$ and $V_3 \leftarrow c$. This is because if any of these values were not pruned, $V_2 \leftarrow a$ would still have support and would not be pruned. Therefore, the conflict set has to include the reason for these prunings as well. In fact, it should be the union of the conflict sets of the pruned values that appear at least once in any of its supporting tuples for the constraint C_{V_1,V_2,V_3}. This knowledge, however, requires time exponential in the arity of the constraint to figure out. [1] Using the union of the conflict sets of the rest of the pruned values in the variables of the constraint can produce less powerful conflict sets, but it is much easier to compute.

When the search reaches a leaf, it means that it has reached a variable that has had its domain wiped out by pruning done at previous levels. It can then use the nogood that it has learned to jump back further than the previous level. [2]

These two algorithms propagate the nogoods that they have discovered to previous levels in the same way as CBJ: by unioning them into the conflict set of the variable they jump back to. This means that they are also able to jumpback from internal nodes.

2.4.2 CFFC, CFGAC

Similar to FCCBJ and GACCBJ, CFFC and CFGAC [1] are versions of FC and GAC that use constraint filtered unioning of nogoods.

[1] Actually, this knowledge is available when using GAC4, which explicitly stores all the supports for every value. It still requires time exponential in the arity of the constraint to compute, however (albeit only at the beginning of the search) and also requires a lot of space, which can be limiting for larger CSPs.

[2] Actually, the reason that a variable has had its domain wiped out has to contain the assignment made at the previous level. Therefore, neither algorithm will jump back more than 1 level from a leaf. It is possible, however, to jump back more than 1 level from internal nodes.

The nogoods that these algorithms learn from constraint propagation are the same that FCCBJ and GACCBJ learn. [3] They differ only in the backtracking algorithm, where the CF versions perform constraint filtered unioning of the per-value nogoods.

2.4.3 A unified view

We can now group the algorithms discussed according to the nogoods that they learn because of constraint propagation and during the search.

An algorithm can learn these types of nogoods because of constraint propagation:

- None
- Nogoods from forward checking
- Nogoods from enforcing GAC

Similarly, it can learn these types of nogoods during search

- BJ-style nogoods $(1 \ldots l)$
- Nogoods resulting from the union of a set of nogoods covering the domain of variable
- Constraint filtered unioning of nogoods

This way, we can create a grouping of the algorithms as shown in figure 2.1.

Each problem might produce more powerful nogoods in only one of the dimensions. It is worthwhile to explore different ways of solving it by trying to do more work in one of these dimensions or balancing them.

[3] So much so, that the pruning routines used in the implementation discussed in chapter 3 are the same for FCCBJ and CFFC and for GACCBJ and CFGAC.

Figure 2.1 Classification of algorithms by level of nogoods learned

	Nogoods from search		
Nogoods from constraint propagation	BJ	CBJ	vsCBJ
None	BJ	CBJ	vsCBJ
FC	FC	FCCBJ	CFFC
GAC	GAC	GACCBJ	CFGAC

2.5 Heuristics

The backtracking algorithms discussed in previous sections all center around the notion of exploring a search tree and using techniques to minimize its size.

One factor that affects the size of the search tree and that these algorithms do not address is the order in which the variables of the problem are instantiated. The only constraints that they place on which variable should be instantiated next are

- If a variable has had its domain wiped out, it should be selected next for instantiation. When a domain is wiped out, it means that no solution can exist in the subtree below the current node. Therefore, there is no reason to search it anymore. In addition, we need to select the wiped out variable for the next level, so that an appropriate jumpback level can be computed.

- If a variable has been reduced to a singleton variable, it should be instantiated next. This is just an optimization.[4]

By finding a better order in which variables are instantiated the algorithm can save an exponential amount of time in learning some nogoods.

There are two main strategies for selecting the next variable. We can either use a static ordering or a dynamic ordering.

[4]It can however be critical to the performance of the algorithm, like in [11].

The static ordering can result from processing the problem before the search has begun.

A dynamic ordering, on the other hand, examines the state of the problem each time if tries to instantiate a new variable and then uses a heuristic to select which one to instantiate. Using a dynamic variable ordering can have dramatic effects on the efficiency of problem solving.

Minimum Remaining Values The most commonly used heuristic is currently the "minimum remaining values" heuristic, or `dom` [17]. Aside from the constraints mentioned earlier, the variable it chooses to instantiate next is the one with the smallest remaining domain size. This heuristic attempts to minimize the size of the search tree below the current node.

Moreover, we hope that by instantiating variables with smaller domain sizes, we can cause more pruning when using a constraint propagation algorithm.

Minimum Remaining Values with tie-breaking by degree The drawback of `dom` is that it treats the constraint hypergraph as a complete graph. This means that in a sparse graph, the next variable will be chosen arbitrarily among those that have an equal (minimum) domain size, with no regard to the connectivity of the variable. This is not desirable, because even though two variables can have the same domain size, when one of them is more constrained that the other, it can potentially cause much more pruning when we make an assignment to it.

Therefore, we can create a new heuristic, MRV with tie breaking by degree, or `dom+deg` [9]. The *degree* of a variable is number of active constraints (i.e. constraints that have not yet been fully instantiated) on that variable. MRV with tie breaking chooses the next variable among those with minimum domain size and among those and selects the one with maximum degree.

Maximum degree When the constraint graph is sparse, the size of the variables' domains can be less important than their degree in choosing the next variable. So, we can use the heuristic **deg** [12] for such CSPs.

Ratio of domain size over degree Bessière and Régin [7] have observed that depending on how constrained a CSP is, different heuristics perform optimally. Specifically, when the constraint graph (or hypergraph) is sparse (small number of constraints), the heuristic **deg** performs better than either **dom** or **dom+deg**. On the other hand, when the constraint graph is dense, **dom** and **dom+deg** perform better.

They proposed that a better heuristic to use would be **dom/deg**, where the variable chosen for instantiation next would be the one with the minimum ratio $domainsize/degree$.

In experiments they performed, using random CSPs, **dom/deg** performed at least as well as either of the other heuristics, but never significantly better than the second best. This indicates that although this heuristic does not produce better results that what was possible with the other heuristics, it does provide a way to get the best results without having to make a choice.

It should be noted, however, that their results only apply to randomly generated CSPs. They may not be applicable to other classes of problems.

2.5.1 Why heuristics work (or not)

The reason why each heuristic works for a specific problem has not been determined. It has been suggested that the reason that **dom** in particular works is that it tries to fail in higher levels of the search tree. It has been proven, however, that this is not the case [33], as heuristics that try to fail early generate search trees with many branches, which slow down the search.

Instead, it seems that a heuristic should try to achieve a balance between failing early and generating trees with few branches. How to measure the difference and estimate the

quality of a heuristic without actually solving the problem is a problem that has not been addressed yet.

Finally, it should be noted that finding the optimal variable ordering for a problem is itself an NP-hard problem, therefore the closest we can get is using heuristic approximations [32].

Chapter 3

Implementation

3.1 Basic Infrastructure

All algorithms discussed in the previous chapter can be viewed as specializations of the algorithm in figure 3.1.

This is a recursive algorithm. It is called by the user program for level 1. In line 2, it goes through all the uninstantiated variables and uses a heuristic to choose one.

In line 3, it checks if no uninstantiated variables remain, in which case it has found a solution, which it processes in line 4.

In lines 6-17, it tries to assign each value in the current domain of the selected variable and then recursively calls itself for the next level. This requires going through all the unpruned values of the current variable and assigning them to it.

In line 7 it makes the assignment. In line 8 it makes sure that this assignment is consistent with the rest of the assignments made so far. In effect, it has to go through all the constraints in which this variable participates. For each constraint that has become fully instantiated, it checks that it is satisfied.

If the assignment is consistent, then it does any constraint propagation needed in line 9. Constraint propagation generally has to go over all the constraints that the variable

Figure 3.1 An abstraction of the algorithms discussed

```
 1 Algorithm genericBT(level)
 2     Choose next variable V
 3     if no variables left then
 4        solution found
 5     endif
 6     for each value v in the domain of V
 7        make assignment V ← v
 8        if assignment is consistent then
 9           do constraint propagation
10           backtracklevel = genericBT(level + 1)
11           undo current assignment
12           if backtracklevel < level then
13              return backtracklevel
14           endif
15        endif
16     endfor
17     unselect( V )
18
19     backtracklevel = computebacktracklevel(level)
20     updateconflictsets_atlevels(backtracklevel, level − 1)
21     return backtracklevel
22 end genericBT
```

participates in. For each constraint, it goes over the current domain of future variables and determines which values no longer have support on the domains of the rest of the variables. Forward checking will only do this for constraints with only one uninstantiated variable, while GAC will do that for all constraints and will cascade any removals.

Only after these steps does the algorithm call itself recursively in line 10. The recursive call will either return the current *level* to indicate that the search should continue in this level or something less than *level* to either signal a jump back or termination of the search (when $backtracklevel == 0$).

Undoing the current assignment in line 11 means that the values that have been pruned because of it need to be restored to their original domains. Therefore, the algo-

rithm also needs to keep track of which values have been pruned at each level.

Lines 19-20 use the nogoods discovered during search to determine to what level the program should jump back to and to update the conflict sets of the values assigned to variables at intermediate levels. This involves determining the maximum level in a conflict set (line 19) and unioning the conflict sets of values at the current level and levels between the current and the jumpback level.

Note that the actual implementation of some parts of this abstract algorithm (lines 8, 9, 19 and 20) may be empty for some of the algorithms. For example, the FC and GAC variants only have consistent values in the current domains of future variables and therefore do not need to do any consistency checks. Similarly, algorithms that do not maintain conflict sets - and therefore only step back, as opposed to jumping back - do not need to do anything for lines 19 and 20, other than:

```
backtracklevel = level - 1
return backtracklevel
```

To implement these algorithms we want to provide representations for each of the following problem elements, as well as methods for manipulating those:

- Variables
- Variable domains
- Conflict sets
- Constraints of arbitrary arity.

Moreover, we will discuss other aspects of the algorithms that need to be provided for by our implementation.

3.1.1 Variables

Variables are objects of `class Var`. Each object contains information about the variable's current domain size and degree, as well as the values in its initial and current domain. They also contain information about the membership of this variable in the constraints of the problem. Internally, the variable objects are identified by a numeric id. In the graph coloring problem, for example, each vertex would correspond to a variable, but would be referenced by its id.

The framework maintains a global array of such objects, called `theVars`. This allows constant time access to the variables by indexing.

In addition, there is an array of uninstantiated variables, which is initialized to contain all the variables before the solver begins. This array allows us to iterate over the uninstantiated variables only, instead of iterating over all of them and skipping those that are already instantiated. This can save time, especially at deeper levels of the tree.

The relevant methods for the above manipulation are provided by the methods `Solver::selectNxtVar` (which corresponds to line 2 of figure 3.1) and `unselectVar` (line 13 of figure 3.1).

Variable domains

The generic algorithm discussed earlier deals with variable domains in the following ways:

- It iterates over all the values of a variable, when manipulating conflict sets (line 16).
- It iterates over all the unpruned values of the variable (lines 5-13). It is preferable if it does this without having to go through and discard those that are pruned.
- The pruning routines (line 8) need to remove values from anywhere on the list of unpruned values in constant time.

- The backtracking part of the algorithm (line 10) needs to add values back to their domains.

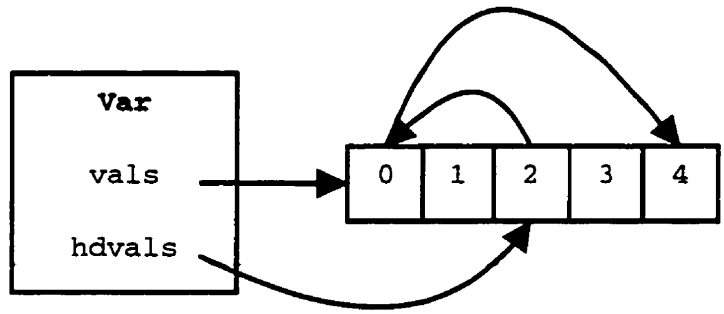
We use an object of `class Val` to represent a value of a variable.

Internally, values are identified by their numeric id. The user part of the program is responsible for treating the numeric id as the corresponding value in the problem domain.

A value object also knows whether it has been pruned and at what level, what its conflict set is and what its variable is.

For each variable, its values are kept in an array indexed by value id, which is simply its numeric value. In addition, each object in the array contains `next` and `prev` pointers to link these objects in a list, as shown in figure 3.2. Only unpruned values remain in this list. This dual structure is used to facilitate the types of access that the algorithms need over the domain of a variable, as mentioned above.

Figure 3.2 Storing the values in a variable's domain



In addition, pruned values are kept in lists, one per level. This is to facilitate adding them back to their domains, when we undo the assignments that pruned them.

Access to the unpruned values of a variable `v` is provided by the container `ValListOfVar(v)`. Similarly, the array of all values is accessed using `AllValListOfVar(v)`.

Values get pruned using the method `Undo::removeVal(val, level)`, which removes `val` from the domain of its variable and places it on the pruned values list for that level.

When backtracking, the method `Undo::restoreVals(level)` restores all the variables pruned at a `level` to their original domains.

Conflict sets We also maintain conflict sets for each value. This is used by the intelligent backtracking algorithms.[1]

The following operations are performed on conflict sets: deriving a conflict set from a constraint (lines 7,8); unioning two conflict sets during backtracking (line 16); and figuring out the maximum level that an assignment in a conflict set was made (line 15).

These conflict sets are maintained as sorted linked lists of ranges of levels. Each node of this list is a `class Cfcell`. It has members `hi` and `lo`. So, the conflict set 1,2,4,5,6,9 would be represented as a list of 3 nodes: [1,2],[4,6],[9,9]. Lists of this type can be easily merged, in linear time. Moreover, we can easily figure out the jumpback level for an intelligent backtracking algorithm, by examining the `hi` member of the last node of the list. To make this operation constant time, we also store a pointer to the last node of the list.

This class, besides the data members that it keeps, also provides convenience methods that perform unioning or constraint filtered unioning of the conflict sets.

3.1.2 Constraints

Conceptually, a constraint C over k variables is simply a set of tuples of length k, such that each tuple is an assignment to the k variables constrained by C. Each tuple is an

[1] We do not maintain a conflict set for the variables, despite the fact that they are needed by the algorithms that learn CBJ-style nogoods during search. This is because the conflict set for the variable is implicitly the union of the conflict sets of all its values. We do not suffer a loss of efficiency for not maintaining a conflict set for a variable, because either way the algorithm learns conflict sets for a value and unions them into the variable's conflict set. There is only a space inefficiency, which cannot be avoided, since we also implement algorithms that learn value specific nogoods.

assignment that satisfies C. The size of the set can be exponential in k. Therefore, the only useful operations that can be performed with a constraint are:

- query it on whether it constrains a variable
- given an assignment to the variables it constrains, find out whether it satisfies the constraint.

The first item can further be refined to checking whether a single variable is constrained and whether a pair of variables is constrained.

In general, constraints are accessed in loops of the following form:

```
Algorithm constraintAccess(Var v)
   foreach constraint C
      if C constrains v
         A = CreateAssignment(C)
         if A ∈ C
            // do something
end constraintAccess
```

or, alternatively:

```
Algorithm constraintAccess2(Var v1, Var v2)
   foreach constraint C
      if C constrains v1 ∧ C constrains v2
         A = CreateAssignment(C)
         if A ∈ C
            // do something
end constraintAccess2
```

Checking constraint membership

The first operation mentioned actually has two facets. The one is having an object `Var *v` and an object `Cons *c` and checking whether `c` constrains `v`. This is done by simply checking that `c->constrains(v)` is true. The supporting structure is little more than a bit vector.

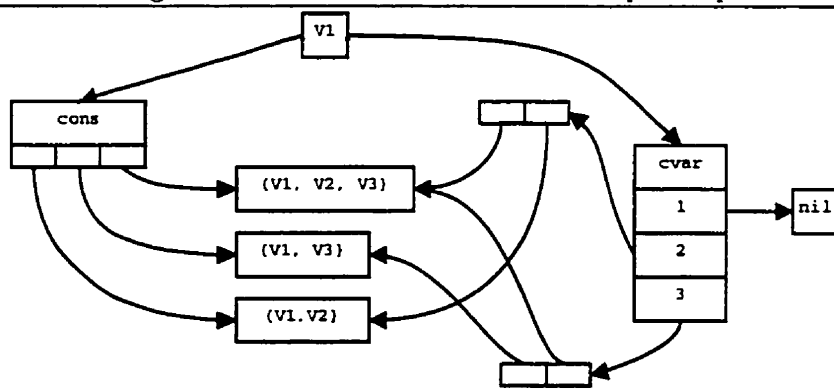
The other facet of this operation is finding, and iterating over, the constraints that constrain a variable or a pair of variables. This is done by accessing the container `ConsListOfVar(v)` or `CvarListOfVar(v1, v2)`, respectively.

These containers are simply wrappers around lists that are maintained for the variable objects. Each such variable object `v` contains a list of all the constrains that constrain this variable. In addition an array of lists is kept. For each variable `v2`, the list at index `v2->id` contains all the constrains the constrain both `v` and `v2`. This arrangement is shown in figure 3.3.

In this figure, we see the data structures maintained for variable V_1. This variable participates in the constraints C_{V_1,V_2,V_3}, C_{V_1,V_2}, C_{V_1,V_3}. The list `cons` contains pointers to each of these constraints. In addition, a list of per-variable constraints is kept in `cvar`. The list for V_1 is always empty, while the list for V_2 points to C_{V_1,V_2,V_3} and C_{V_1,V_2} and the list for V_3 points to C_{V_1,V_2,V_3} and C_{V_1,V_3}. The same structures are kept for each of the other variables, but are not shown in the figure. Pointers from the constraints back to the variables are also kept, but not shown in the figure.

These lists are filled when completing the initialization of the CSP.

Figure 3.3 Associating a variable with the constraints it participates in



Checking constraint satisfaction

The actual reason that constraints exist at all is to check whether or not assignments satisfy them.

However, the representation of a constraint is not easily abstracted. One common abstraction is the extensional representation, where a boolean value is used to indicate whether an assignment satisfies a constraint. For a constraint of arity n, we store these boolean values in an n-dimensional array, where the index for dimension k is the id of the value assigned to the kth variable of the constraint.

This representation, however, can be very space-inefficient for problems with constraints of large arity or large domains, as the space complexity of this structure is $O(n^d)$, where n is the arity of a constraint and d is maximum of the sizes of the variable domains. Not only that, but the construction of this array also has time complexity $O(n^d)$. On the other hand, a procedural representation for the same constraint might have constant initialization time complexity and constant space complexity.

Therefore, we need a way for the user of the framework to provide a procedural representation of their own, or use the extensional representation if this is convenient.

To provide for this, we define a base class `ConsRep`, which abstracts the functionality of the constraint representation. This class provides a virtual function `checkAssgn`, which must be overridden by derived classes to return the truth value of an assignment. Thus, a `ConsRep*` can be an interface to any kind of constraint checking mechanism.

The framework provides two builtin classes, `ExtensionalConsRep` and `LibIntentionalConsRep`. The former provides the $O(n^d)$ extensional representation discussed above. The latter allows the user to define constraint checking functions in a shared library, load this library dynamically at runtime and forward the checking to the functions in the shared library.

Given all that, objects of `class Cons` actually do not do any checking. Instead, they maintain a list of objects of `class ConsRep`. When calling `c->checkAssgn()` for

a constraint c, what it actually does is go through every ConsRep object in its list and ask them whether the assignment satisfies them. It returns the conjunction of the results that it gets.

The reason we use a list of ConsRep objects for each constraint is that the framework supports posting only one constraint over one set of a variables. This is not a limitation on what kinds of problems can be represented, since a constraint that is the conjunction of several constraints over the same variables is equivalent to them (but can cause more pruning when doing GAC propagation). It can, however, be inconvenient for the user to have to manually conjoin two constraints that are logically separate in the model of the CSP she is trying to solve. Therefore, we provide a way for the user to define two logically distinct constraints over the same set of variables and have them automatically conjoined by the framework.

Propagators

The final aspect of constraint checking that we have to address is constraint propagation.

Constraint propagation can be in the form of forward checking, enforcing various forms of arc consistency checking or something more elaborate. It could be argued that whether we do forward checking or arc consistency is actually a matter of the which algorithm we choose (a variant of FC or of GAC). However, a unified view of constraint propagation helps to better structure the code. Moreover, some types of constraints can have a special structure, such that an algorithm can be developed for doing constraint propagation with a time complexity significantly smaller than that of standard constraint propagation. This type of constraint propagation could be used by all types of algorithms.

Consequently, each constraint maintains a list of appropriate propagators. When an algorithm makes an assignment, it can choose to call these propagators and take advantage of their domain knowledge. The difference between vsCBJ, FC and GAC is that the first two will just accept the results of the propagation, while GAC will regard

it as just a more efficient way of enforcing GAC on a constraint.

Like constraint representations, there is no generic way to implement a constraint propagator. The implementation has to be abstracted to a base class `Propagator` that provides the interface for propagation. The derived classes have to provide implementations for virtual functions related to initialization, propagation, backtracking and management of the queue for GAC. Specifically, when doing propagation, the derived class is notified of which values have been pruned, so it can update its internal representation (if any). Each propagator has to return a list of values that it has decided should be pruned and the variables that are affected by this pruning.

Each constraint has a list of associated propagators with it. The reason for having a list, instead of a single propagator is, as with constraint representation, the ability to have logically separate implementations and have the framework take care of conjoining their results, instead of having to manually do it.

Propagator example: the *AllDiff* propagator The AllDiff constraint is a special type of constraint. It dictates that the value of its variables are all different. In a CSP, it can be implemented either by a clique of binary not-equals constraints for all the concerned variables or as a constraint of arity n. Either of these will find the same solutions.

However, when doing GAC propagation, their effects are quite different. Consider the example of an AllDiff constraint among the variables V_1, V_2, V_3 with domains $\{1,2\}, \{1,2\}$ and $\{1,2,3\}$ respectively. Enforcing GAC on the binary not-equals representation will not prune any variables. Doing it on the constraint of arity n however will prune values 1 and 2 from the domain of V_3. The complexity of enforcing GAC on a constraint of arity n, however, has time complexity $O(n^d)$, which is prohibitively expensive.

J.C. Règin [26] has proposed a special purpose propagator for this type of constraint, which allows us to have stronger arc consistency with small complexity. The idea is that

we transform the constraint into a bipartite graph. The vertices in one partition, X, represent variables and the vertices in the other partition, Y, represent values. An edge between two vertices means that the value is in the domain of the variable.

Next, we find a matching on the bipartite graph. A *matching* is a set of edges such that each vertex is adjacent to at most one of the edges comprising the matching. A matching that covers a set of vertices is simply a matching such that all vertices are adjacent to exactly one of the edges in the matching.

Given that, finding an assignment that satisfies the constraint translates to finding a bipartite matching that covers the partition X.

We can use this idea to also do constraint propagation. Specifically, given a matching that covers X, we can decide whether there is any matching that contains a given edge. The conditions for this are that the edge either belongs to the current matching, or to an alternating path of even length, or an alternating cycle.

The complexity for this, for the worst case that the propagator gets called to remove the values of each variable one by one is $O(n^2d^2)$. If we make the crude assumption that the time spent in the propagator is equally divided among the nodes along the path to a leaf of the search tree, then the complexity at each node is $O(nd^2)$. In contrast, the complexity for generic GAC enforcing for this constraint would be $O(dn^d)$ at each node.

The routines described in [26] map exactly to the virtual functions that the class we derive from `Propagator` has to implement. Therefore, the implementation is rather straightforward.

3.2 Algorithms

The implementation of the algorithms discussed is relatively straightforward, once the rest of the framework is in place.

They are simply translations of the pseudocode in figure 3.1, calling functions to do

consistency checking (line 8), constraint propagation (line 9) and conflict set manipulation (lines 19 and 20, as well as inside the functions for consistency checking and constraint propagation).

`CSP::assgnConsistent` checks whether the latest assignment made is consistent with the previous assignments and returns the first constraint that is inconsistent or NULL if the assignment is still consistent.

The following functions do conflict set manipulation:

- `Cfcell::setConsConflict` - sets the conflict set of a value to be the set of levels where assignments where made to the variables of a constraint, excluding the current variable.
- `Cfcell::setConflictSet` - sets the conflict set of a value to be the union of the conflict sets of all the values that are compatible with it in the domain of another variable.
- `Cfcell::setCBJConflictSet` - sets the conflict set of a value to be the union of the conflict sets of all the values in the domain of another variable
- `Cfcell::mergecfs` - unions two conflict sets.

Finally, the functions that need to be called for pruning:

- `FCPrune` - forward checking, without conflict set manipulation
- `FCCfPrune` - forward checking, setting conflict sets for pruned variables.
- `FCCfLookBackPrune` - forward checking, setting the conflict set of the pruned variable to be the union of the conflict sets of all the values that support at the current level.
- `GACPrune` - maintains arc consistency at the current node, using a flag for whether to use conflict sets.

GACPrune `GACPrune` is the only function that is not trivial to implement using the underlying framework.

`GACPrune` has to use propagators to enforce GAC as efficiently as possible. As described in chapter 2, AC-3, the version of AC that we implement, stores pairs $< V, C >$ in a queue. Whenever it processes one such pair, it makes sure that all the values of V are GA consistent with respect to C. Processing a pair involves invoking the propagator for that pair and pruning the values that the propagator decides are GA-inconsistent. After that, it determines which variables have had their domain pruned. For each such variable V', it finds all variables V'' that are constrained with V' via a constraint C'' and inserts every pair $< V'', C'' >$ into the queue. The reasoning for this is that since V' has had its domain pruned, then some of the values of V'' may have lost their support for constraint C''.

However, it is not always desirable to insert every possible pair into the queue. There are two reasons why this is so:

- The user may have requested that GAC is not enforced for some constraints under specific circumstances:
  - Because it is too expensive to do so at that point (for example, we may want to enforce GAC on a 10-ary constraint only when at least 7 of the variables are instantiated. This is equivalent in time complexity to enforcing GAC on a 3-ary constraint)
  - Because there is a cheaper way to do perform the same task (When we use redundant constraints)
- The propagator may examine all the variables of a constraint at once. Such an example is the AllDiff propagator, which needs to have its constraint inserted into the queue only once each time that some of the variables it constrains have their domain pruned.

Therefore, before inserting a pair $< V, C >$ into the queue, the algorithm asks the constraint representation for this if it wants to be inserted into the queue. This is user controllable and corresponds to the first of the items listed above. If this succeeds, it then asks the appropriate propagator if it wants to be inserted into the queue. This is not user controllable, but rather intrinsic to the propagator. For example, a general purpose GAC propagator will need to be inserted into the queue for every pair, but an AllDiff propagator should not be in the queue more than once at any given moment.

Currently, GACPrune uses a queue, but it really only needs a structure that supports the properties of a set:

- Insert an item in the set if it is not already there
- Remove an item from the set

Therefore, we might use a stack or a priority queue instead of a queue. Depending on the problem currently being solved, it can benefit the speed of enforcing GAC.

3.3 Heuristics

Currently, the only heuristics supported by the framework are *DVO* (dynamic variable ordering) heuristics.

The user program selects what kind of DVO heuristic should be used. The framework provides some heuristics, but it is fairly easy to write other ones as needed.

The user program has to call the algorithm like this:

```
fnpt(1, heuristic);
```

Where `heuristic` is an object of a subclass of `class Heuristic` with `operator()` defined. When the algorithm needs to select the next variable to assign. The operator needs to get `Solver::unasgnVars`, which is an array of unassigned variables and return

an index into this array. The user program passes this on to the `selectNxtVar` to do all the bookkeeping work of selecting this variable.

Using objects like this allows us to implement heuristics that are more easily customized to the needs of the specific problem. The obvious alternative, using pointers to functions does not allow this without using non-obvious means of communication between the problem setup code and the heuristic code, such as global variables.

To make this clear, consider the case of two common heuristics: `dom` and `rdom`. The first one simply selects the variable with the smallest remaining domain size, while the second one only considers some of the variables of the problem and applies `dom` to the selection and considering the rest of the variables only after all the first variables have all been assigned to. This can be used in problems where we use auxiliary variables to enforce stronger constraints, but we do not want to initialize those variables earlier than the primary ones.

Implementing `dom` with either method should be trivial. However, we cannot implement `rdom` using pointers to functions, without using a global variable (or global structure) to indicate which variables are considered auxiliary. Using objects, on the other hand, makes this easy, as we simply incorporate this information into the subclass which we use to compute the heuristic.

As a convenience, three heuristics are provided by the framework: `dom+deg`, `lexicographic` and `dom/deg`.

One thing that is not enforced by the framework is the fact that if there are variables whose domains have been wiped out, one of these should be returned by the heuristic. This is required for better performance, since there is no solution in a subtree where one variable has had its domain wiped out. It is however also required for all the constraint propagating algorithms to work correctly. Similarly, if no variable has had a domain wipeout, the heuristic should prefer a variable that has a singleton domain. This is a

matter of performance only,[2] having no implications on the correctness of the algorithms.

[2]As mentioned, this optimization can be critical to the performance for algorithms like Davis-Putnam [11].

Chapter 4

Experiments

4.1 Overview

In this chapter, we will demonstrate how the framework that we described allows us to easily perform experiments to evaluate the performance of algorithms and heuristics on different problems. We will present an automatic method of manipulating a model and potentially improve it when using the GAC family of algorithms. We will test this method in two problems: the optimal golomb ruler problem and the random 3SAT problem. In addition, we will examine the results of [31] on the optimal golomb ruler problem and compare the results obtained.

4.2 Automatic conjunction of constraints

Bessière and Régin suggested in [6] that it is possible to use GAC on conjunctions of constraints to increase pruning and therefore reduce the size of the search tree. They asserted that models in practical application are usually created by identifying constraints as a conjunction of subconstraints and using the subconstraints to model the problem. However, GAC on the subconstraints is not as powerful, in terms of pruning, as GAC on the original constraint.

Bessière and Régin also suggested that it is possible to conjoin constraints that are not semantically related.

In addition, they prove that it is possible to enforce GAC on a conjunction of the constraints in $O(d^{\|\cup_i VarsOf(C_i)\|})$, as opposed to $O(d^{\|VarsOf(C_i)\|})$ for each constraint i.

Finally, they present experimental results to prove that it can be beneficial in terms of cpu time as well as well as number of backtracks performed, even though the benefits in terms of cpu time only exist in harder problems.

In this chapter, we will examine the reason why conjunctions of constraints afford more pruning and develop a heuristic for choosing constraints that should be conjoined.

4.2.1 Why conjoining causes more pruning

The set of assignments that satisfy the conjunction $C_1 \wedge C_2$ of two constraints is the set

$$\left\{\mathcal{A} : \mathcal{A}_{C_1} \in C_1 \wedge \mathcal{A}_{C_2} \in C_2\right\}$$

Where $\mathcal{A}_{C_1}$ is the projection of $\mathcal{A}$ on the $VarsOf(C_1)$. This implies that $\mathcal{A}$ is the union of two satisfying tuples for C_1 and C_2, such that the assignments to their common variables are the same. In other words, not every pair of tuples satisfying the original constraints will satisfy the conjoined constraint.

This means that for a value of a variable to not be pruned when enforcing GAC on the conjoined constraint, we have to find a satisfying tuple for the conjoined constraint, which depends on a stronger condition being satisfied than simply that a satisfying tuple exists for each of the constraints C_1, C_2.

In addition, the condition becomes stronger as the two constraints have more variables in common. Consider the following cases:

- $\|VarsOf(C_1) \cap VarsOf(C_2)\| = 0$

  In this case, a satisfying tuple for $C_1 \wedge C_2$ is simply the union of any supporting tuple of C_1 and C_2. This is no stronger than GAC on C_1 and C_2 individually.

- $\| VarsOf(C_1) \cap VarsOf(C_2) \| = 1$

  In this case, when checking whether the assignment $V \leftarrow x$, where V is the common variable, is GA-consistent, all the GAC algorithm has to do is find a supporting tuple for $V \leftarrow x$ in both C_1 and C_2, without any additional constraints. This is what GAC on the original constraints does as well, so this case is also no stronger than GAC on the original constraints.

- $\| VarsOf(C_1) \cap VarsOf(C_2) \| > 1$

  In this case, the tuples that satisfy the conjunction are no longer simply the union of any two satisfying tuples of the original constraints. Instead, the number of tuples that can be unioned to create a satisfying tuple for the conjunctive constraint becomes smaller, as they have to have more common assignments. The relative strength of GAC in the conjunction increases as well.

This reasoning can be extended to the case when we conjoin n constraints, by applying pairwise conjunctions.

4.2.2 When to conjoin constraints

Based on the above observations, we can develop a heuristic method to determine which sets of constraints should be conjoined and create the set CS of all such sets:

1. Initialize the set CS to be the set $\{\{C_i\} \forall i\}$, which means that each constraint is placed in a set by itself.

2. If $C_i \in CS \wedge C_j \in CS \wedge \| VarsOf(C_i \cup C_j) \| - \max_{C \in C_i \cup C_j} \| VarsOf(C) \| \leq M$, then remove C_i and C_j from CS and insert $C_i \cup C_j$. $C_i \cup C_j$ is the conjunction of the constraints in C_i and C_j. In other words, if the arity of the constraint that results from conjoining the constraints in C_i and C_j is no more than M greater than the

arity of any of the *original* constraints, then conjoin all the constraints in C_i and C_j.

3. Repeat 2 until no more conjunctions can be made.

The parameter M limits the constraints that can be conjoined. When $M = 0$, two constraints will only be conjoined when one is over a subset of the variables that the other covers. In this case, the complexity of GAC does not increase at all, since the constraints of higher arity are not created, while the advantages of conjoining constraints remain.

As M increases, the complexity of GAC increases as well. It depends on the problem at hand what the value of M should be. In the following sections, we will examine two such problems.

4.3 Optimal Golomb Rulers

A golomb ruler is a set of non negative integers, $m_0, ..., m_n$, called *marks*, such that the distances $m_i - m_j, i > j, \forall i, j$ are distinct.

Moreover, the *length of a golomb ruler* is defined as $max(m_i - m_j), \forall i, j$. The first mark is typically 0, so the length of the ruler is actually $max(m_i)\forall i$. An *optimal* golomb ruler is one such that no ruler of smaller length with the same number of marks exists.

Smith et al. in [31] have studied various alternative representations, heuristics and extra constraints that one can add to make solving the problem more efficient. We will try to reproduce some of their results here.

4.3.1 Representations

This problem can be represented as a Constraint Satisfaction Problem, by posting a constraint $m_i - m_j \neq m_k - m_l, \forall i > j, k > l$. This also implies that we post ternary constraints, for the special case when $i = l$. To limit the number of symmetric solutions,

we also post the constraints $m_i < m_j, \forall i < j$ and also $m_2 - m_1 < m_n - m_{n-1}$ to eliminate symmetric solutions. Using this representation, we post the following constraints over every set of 4 variables:

$$
\begin{aligned}
x_1 - x_0 &\neq x_3 - x_2 \\
x_2 - x_0 &\neq x_3 - x_1 \\
x_3 - x_0 &\neq x_2 - x_1 \\
x_2 - x_1 &\neq x_1 - x_0 \\
x_3 - x_1 &\neq x_1 - x_0 \\
x_3 - x_2 &\neq x_2 - x_0 \\
x_3 - x_2 &\neq x_2 - x_1
\end{aligned}
$$

The first 3 constraints are quaternary constraints, while the other 4 are ternary. It should be noted that these are not all the constraints that the formula $m_i - m_j \neq m_k - m_l, \forall i > j, k > l$ implies. There are many constraints that are essentially duplicated (e.g $x_1 - x_0 \neq x_3 - x_2$ is equivalent to $x_3 - x_2 \neq x_1 - x_0$) or otherwise redundant ($x_2 - x_0 \neq x_1 - x_0$ is implied by the fact that $x_0 < x_1 < x_2$).

Smith et al. [31] assert that the obvious representation is not optimal. They propose an alternate representation, where each mark is a variable in the CSP, as well as $m(m-1)/2$ auxiliary variables. For every two mark variables, an auxiliary variable is used to represent their difference and a constraint to that effect is posted among these three variables. In addition, we post a not-equals constraint between every two auxiliary variables. The authors prove that this representation will generate more pruning than the one with quaternary constraints in every case except when the quaternary constraint is actually a ternary constraint of the form C_{V_i,V_j,V_j,V_k}.

This representation can be further improved if a single alldiff constraint is posted over

all the auxiliary variables instead of a clique of binary constraints. [1]

Moreover, different dynamic variable ordering strategies are compared for the best representation, comparing a static lexicographic ordering, the `dom` heuristic and the `rdom` heuristic (`dom` restricted to the mark variables).

4.3.2 Results

Since we use different software, as well as hardware, from what Smith et al. used, the results are not directly comparable. It should also be noted that they used an optimization feature to discover the optimal golomb ruler, something which is not currently provided by our framework. We can, however, reproduce their comparative results.

Comparing representations

In addition to the representations studied by Smith et al., we study an additional representation, which is automatically derived from the one using quaternary constraints by applying the heuristic discussed earlier with the parameter M set to $M = 0$.

We count the number of recursive calls made to find a ruler of the given length (or prove that none exists). The results are shown in figure 4.1. In this table, the first two columns show the size of the problem tested, the first being the number of marks and the second the length of the ruler. After that, we have two columns for each of the four representations tested: the original one using quaternary constraints, the conjunctive one which also uses quaternary constraints and the two representations using ternary constraints Smith et al proposed. The first column for each representation shows the number of backtracks performed by the solver and the second shows the cpu time it used to solve the corresponding problem.

Note that the representation using conjunctive quaternary constraints fares much

[1] Régin [26] discusses the advantage of a single alldiff constraint over a clique of binary not-equals constraints as well as a way to efficiently implement it.

better than the original quaternary representation. The resulting single conjunctive constraint generates much more pruning when doing GAC propagation.

Figure 4.1 Backtracks performed and cpu time to find (F) a golomb ruler of a given size and prove (P) its optimality. "-" indicates that the solver was unable to find a solution after 2^{31} constraint checks

Problem		Quaternary		Conj Quat		Ternary			
						Not-Equals		AllDiff	
size	length								
7	25 (F)	95	0.14	90	0.06	90	0.44	89	0.45
7	24 (P)	988	1.62	894	0.51	885	5.78	759	4.3
8	34 (F)	574	1.33	492	0.45	443	5.59	395	2.79
8	33 (P)	7791	27.51	7131	7.99	6787	106.32	5070	42.88
9	44 (F)	5581	26.51	4920	7.75	4325	118.4	3544	40.13
9	43 (P)	57545	403.28	52868	117.16	51331	1711.24	32781	429.22
10	55 (F)	40141	360.64	36666	107.24	33273	1968.69	22427	389.55
10	54 (P)	-	-	373375	1581.53	348930	23852.78	176150	3797.4
11	72 (F)	-	-	234069	1058.47	203041	23325.09	133798	3110.34
11	71 (P)	-	-	-	-	-	-	-	-

We notice that the conjunctive quaternary representation is approximately as powerful as the ternary+not-equals representation, while enjoying much better times due to the simplicity and smaller number of constraints [2].

We can also see that the representation that used the `AllDiff`propagator does not perform as well as the one in the commercial product ILOG Solver that was used in [31], but its relative performance compared to the representation using a clique of not-equals constraints is what is expected.

[2]Actually, both representations have $O(m^4)$ constraints, but the representation with conjunctive quaternary representations has $O(m)$ variables as opposed to $O(m^2)$ variables

In general, the higher times produced here can be attributed to the implementation of the GAC-enforcing algorithm, which is based on AC-3. In contrast, ILOG Solver uses an algorithm based on AC-7, which scales better than AC-3.

In addition to this test, we performed a test using different levels of conjoining for the quaternary representation, shown in figure 4.2.

Figure 4.2 Backtracks and cpu time to find (F) a golomb ruler of a given size or prove (P) its optimality. "-" indicates that the solver was unable to find a solution after reaching 10^5 leafs

Problem		Quat + Tern		Conj Quat + Tern		Conj Quat only	
size	goal						
7	F	95	0.14	95	0.07	90	0.06
7	P	988	1.62	988	0.81	894	0.51
8	F	574	1.33	574	0.64	492	0.45
8	P	7791	27.51	7791	13.35	7131	7.99
9	F	5581	26.51	5581	12.53	4920	7.75
9	P	57545	403.28	57545	190.16	52868	117.16
10	F	40141	360.64	40141	167.45	36666	107.24
10	P	-	-	-	-	-	-

The first column in that table shows the original quaternary representation, while the last columns show the conjoined representations. In the second column, we show the results for an intermediate representation (which was not generated automatically), which consists of a conjunction of the quaternary constraints, but without the ternary constraints.

The results are not surprising, other than the fact that the first two representations perform the exact same number of backtracks in every case. This can be explained by the fact that the 3 quaternary constraints posted every set of 4 variables are actually

equivalent. The constraints are:

$$x_1 - x_0 \neq x_3 - x_2 \tag{4.1}$$

$$x_2 - x_0 \neq x_3 - x_1 \tag{4.2}$$

$$x_3 - x_0 \neq x_2 - x_1 \tag{4.3}$$

It turns out that we only need to post the first of these constraints, as the others are implied. In particular, we can add $x_2 - x_1$ to 4.1 to get 4.2, while 4.3 is implied by the order constraints between the variables ($x_i < x_j, i < j$).

This demonstrates the fact that conjoining constraints is not only an automatic way to improve a model, but also good for analysis. Aspects of the model which were not previously clear can be revealed by analyzing the behavior of the derived models.

From this point on, all results will report on the conjoined quaternary representation instead of the original quaternary representation.

Comparing variable ordering strategies

We will compare two orderings for this problem: a static lexicographic and the `dom+deg` heuristic.

Surprisingly, the lexicographic ordering turns out to be the best strategy for this problem, as shown in figure 4.3.

We actually see a different behavior for the two heuristics, depending on whether we use the representation with quaternary constraints or the one with ternary constraints and the Alldiff propagator.

The lexicographic ordering gives better results in all cases when using ternary constraints. The reason for this is that selecting the variable with the minimum remaining domain size can often select one of the auxiliary variables. The value of these variables, however, depends entirely on the value of the mark variables it is constrained with. When

Figure 4.3 Number of backtracks (leafs) to find (F) a golomb ruler of a given size and prove (P) its optimality using different DVO heuristics

Problem		Quaternary		Ternary+Alldiff	
size	length	`dom+deg`	`lexicographic`	`dom+deg`	`lexicographic`
7	25 (F)	90	90	194	89
7	24 (P)	824	894	706	759
8	34 (F)	491	492	13340	395
8	33 (P)	6231	7131	34264	5070
9	44 (F)	4947	4920	3147390	3544
9	43 (P)	42844	52868	3114791	32781
10	55 (F)	30962	36666	1035795	22427
10	54 (P)	280822	373375	1221568	176150
11	72 (F)	183960	234069	275181	133798
11	71 (F)	-	-	-	-

none of the other two variables have been assigned, it will cause minimal pruning. Because the total number of auxiliary variables is relatively high ($O(m^2)$), there is a high probability that this will happen often. Therefore, the system spends much time doing work that does not gain anything.

Instead, the lexicographic ordering makes sure that the mark variables are assigned first.

We also notice that when using quaternary constraints, the `dom+deg` heuristic does not significantly improve the performance of the solver. This can be attributed to the nature of the problem and the representation. In the original definition of the problem, all mark variables are equivalent. Therefore, there are many symmetric solutions. When solving this problem, the `dom+deg` heuristic would be expected to perform much better than the `lexicographic` heuristic.

However, we do not want to report all symmetric solutions. Therefore, we impose the following constraints to break symmetries:

- $m_i < m_j, i < j$. This constraint eliminates an exponential number of symmetric solutions.
- $m_2 - m_1 < m_k - m_{k-1}$, where k is the length of the ruler. This constraint only eliminates one symmetric solution.

Of these, the first type of constraint is the one that makes the difference. Because of it, all mark variables are no longer equivalent. In fact, the optimal ordering should almost always be very close to the lexicographic ordering of the variables. This is because, all other things being equal, as more variables become instantiated in lexicographic order, the tighter the constraints become for the next variables and therefore more values get pruned from domains.

Comparing algorithms

The authors of [31] used ILOG's Solver for their experiments. This solver uses AC-7 to enforce GAC, instead of AC-3 used by our implementation. This causes their results to show GAC stronger in terms of time than ours. However, they do not consider FC and conflict filtered backtracking algorithms at all.

The table in figure 4.4 shows that using CFFC instead of GAC type algorithms can dramatically improve the performance of finding a solution. Similarly, CFGAC performs better that GAC, both in terms of time and number of recursive calls.

From this table, we can see that CFGAC is better than GAC at finding a solution both in the number of backtracks performed, as well as in cpu time. When proving the optimality of a given ruler, it is better in terms of recursive calls and comparable in terms of cpu time. This means that maintaining the conflict sets is an overhead that pays off when finding the optimal ruler but not when proving that it is optimal.

Figure 4.4 Number of backtracks and cpu time to find (F) a golomb ruler of a given size and prove (P) its optimality using different algorithms

Problem		GAC		CFGAC		CFFC	
size	length						
7	25	90	0.06	69	0.06	129	0.01
7	24	894	0.51	825	0.53	2648	0.16
8	34	492	0.45	297	0.36	484	0.05
8	33	7,131	7.99	6,534	8.13	18,751	1.88
9	44	4,947	7.75	3,368	6.05	5,209	0.83
9	43	52,868	117.16	48,329	116.43	12,948	19.97
10	55	36,666	107.24	26,712	83.63	40,483	8.00
10	54	373,375	1581.53	344,876	1555.58	928,159	207.06
11	72	234,069	1058.47	145,507	701.29	187,127	59.6
11	71	-	-	-	-	15,963,914	4958.79

CFFC is significantly better than both GAC and CFGAC in terms of time. As expected, the number of backtracks performed is higher, but only by a factor of less than 2 when finding the optimal ruler and less than an a factor of 3 when proving optimality. The factor gets better as the size of the problem increases and it reaches fewer leaves than GAC (but more than CFGAC) when finding the optimal ruler with 11 marks. The fact that it does so much less work at each node makes it faster overall, usually around an order of magnitude better.

What this shows is that the conflict sets discovered by CFFC allow the algorithm to jump approximately back to the point where GAC would reach a leaf. Not only that, but after that point, any jumps further back are almost the same ones performed by CFGAC. In other words, CFFC explores approximately the same portion of the search space as CFGAC, only it has to reach a deeper level before performing each backtrack.

The size, however, of the subtree that CFFC explores and CFGAC does not is linear in size, as opposed to exponential for the general case.

4.4 Random 3-SAT instances

We will use the CSP representation to solve random instances of the 3-SAT problem. Given N and C, the number of variables and the number of clauses, we generate instances by randomly selecting 3 literals out of the $2N$ (positive and negative) literals, discarding tautologies. We then construct the clause $l_1 \vee l_2 \vee l_3$.

Using CSPs to solve SAT problems is not optimal, since CSP search algorithms do not take advantage of the special structure of SAT constraints (i.e. 3-clauses). SAT can, however, be used as an interesting benchmark to compare CSP algorithms. It has been shown [1] that CFFC is the best algorithm for solving this type of problems, if we use a CSP solver.

Instead, we will focus on GAC and the use of conjunctive constraints.

4.4.1 Representations

The first representation we use is one where we simply post a ternary constraint for each clause.

The second representation is derived from the first one, by applying the heuristic described with the parameter $M = 1$. The resulting model contains a mix of ternary and quaternary constraints.

4.4.2 Results

We ran the Solver for both representations at the crossover point $c/n = 4.26$ [21] for a number of variables ranging from 60 to 100 with a step of 10. We count the average

cpu time and average number of backtracks performed by the solver, using the GAC algorithm. The results are shown in figure 4.5.

Figure 4.5 Average number of backtracks and cpu time to prove whether a problem is satisfiable or not. The last column indicates the percentage of instances where the solver performed better if constraints were conjoined

		avg leafs		avg time		
# Variables	# Instances	original	w/conj	original	w/conj	perc
60	100	674.33	522.71	0.9951	0.8485	71
70	100	1637.36	1300.21	2.8882	2.4676	74
80	100	3504.36	2888.33	7.204	6.3182	74
90	100	9204.49	7085.21	21.7589	17.8116	75
100	100	19573.2	14434.2	52.5704	40.7247	83

It is worthwhile noting in this case that there were instances where the solver actually performed more backtracks when using conjoined constraints than it did in the original problem. This anomaly can be attributed to the fact that 3-SAT has a special structure, which is not accounted for in the DVO heuristic used. Therefore, even though the conjunctive constraints cause more pruning, they end up making the search slower. This anomaly can probably be eliminated by using one of the heuristics that have been developed for SAT solvers.

In the instances where the solver performed fewer backtracks with conjunctive constraints, the cpu time used is at most 10% worse than the time used to solve the problem using the original model. This shows that the overhead of performing GAC on the conjoined constraints is alleviated by the extra pruning that it causes. The refined results are shown in figure 4.6.

Figure 4.6 Average number of backtracks and cpu time to prove whether a problem is satisfiable or not. The last column indicates the percentage of instances where the solver performed better if constraints were conjoined. Only instances for which conjoining constraints did not interfere with the behavior of the DVO heuristic are counted

		avg leafs		avg time		
# Variables	# Instances	original	w/conj	original	w/conj	perc
60	79	1602.62	540.81	2.35886	0.879747	88.6076
70	78	3890.79	1322.47	6.84756	2.50756	93.5897
80	82	7829.66	2706.33	16.0824	5.94561	90.2439
90	79	21926.1	7214.34	51.7696	18.1687	94.9367
100	85	44843.2	15398.1	120.359	43.482	97.6471

4.5 Conclusions

In this chapter, we demonstrated the ability of the framework described in chapter 3 to support experiments in the field of CSPs. We tested it on two widely used benchmark problems, the optimal golomb ruler problem and the random 3SAT problem.

In addition, we proposed a simple technique to automatically improve the efficiency of a model when using the GAC algorithm. We demonstrated the ability of the framework to support this new technique and used it to show that it can indeed produce better models. We did note possible problems that it can introduce by interfering with the DVO heuristic used, but overall the results were encouraging.

Chapter 5

Conclusions

In this thesis, we have examined existing CSP search algorithms in a consistent manner which allows for their implementation in a common framework. Specifically, we found the algorithms examined differ only in the level of constraint propagation that they perform and in the granularity of the conflict sets that they maintain. This enables us to treat them uniformly, at least from the point of view of what they require to work and how to provide that.

Similarly, we provided an overview of commonly used variable ordering heuristics and showed how they can be treated consistently.

The CSP solver that was implemented based on these findings is a general-purpose, flexible solver that not only can be extended to accommodate new algorithms and new variable ordering heuristics, but also allows for experimenting with models, special purpose propagators and extensions of the CSP model.

5.1 Future Work

The framework that was developed for this thesis can be used to facilitate research in the following areas:

- Integration of CSP algorithms with other search techniques. One example of this might be integrating a CSP solver with integer programming algorithms.

- Modeling. We have already demonstrated (in chapter 4) that it is possible to use the framework developed here not only to test alternative representations to solve the same problem, but also to automatically aply transformations to an existing model as a way to improve solver performance. Not only that, but pre-existing known transformations (e.g. from a non-binary problem to a binary one, using either the dual [13] or the hidden variable [27] transformation) are hard to apply manually. Instead, it is easier to define a model and have a routine do the transformation. This allows us to perform experiments on a wide array of problems to test the effectiveness of different modeling techniques.

- Search algorithms. We have implemented here several of the most popular algorithms used for solving CSPs and implementing new ones in an efficient manner should be straightforward.

- Dynamic variable and value ordering heuristics. As mentioned, little is understood about the implications of dynamic variable ordering heuristics on the performance of the search algorithms discussed. It would be interesting to study the behavior of known heuristics in problems other than random CSPs [7, 33, 16].

- Extension of the CSP model. It has been proposed to extend the CSP model to allow the use of hard and soft constraints (e.g VSCP [29]) or other approaches to optimization. This field, however, has not been explored in depth.

Appendix A

Availability

The result of the work described in this thesis can be obtained online at

`http://www.cs.toronto.edu/~gkatsi/efc.tar.gz`

Bibliography

[1] Fahiem Bacchus. Extending forward checking. In *Principles and Practice of Constraint Programming*, pages 35–51, 2000.

[2] H. Bennaceur. The satisfiability problem regarded as a constraint satisfaction problem. In *Proceedings of the 12th ECAI*, pages 155–159, 1996.

[3] Michael E. Bergen, Peter van Beek, and Tom Carchrae. Constraint-based vehicle assembly line sequencing. In *Proceedings of the 14th Canadian Conference on Artificial Intelligence*, pages 88–99, Ottawa, Ontario, 2001.

[4] C. Bessière. Arc-consistency and arc-consistency again. In *Artificial Intelligence 65*, pages 179–190, 1994.

[5] C. Bessière, E.C. Freuder, and J.C. Régin. Using inference to reduce arc-consistency computation. In *Proceeding of the 14th IJCAI*, Montréal, Canada, 1995.

[6] C. Bessière and J.-C. Régin. Local consistency on conjunctions of constraints. In *Proceedings of the ECAI'98 Workshop on Non-binary constraints*, pages 53–59, Brighton, UK, 1998.

[7] Christian Bessière and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Principles and Practice of Constraint Programming*, pages 61–75, 1996.

[8] J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Comm. ACM*, 18:651–656, 1975.

[9] D. Brélaz. New methods to color the vertices of a graph. In *Communications of the ACM, 22*, pages 251–256. 1979.

[10] P. Burke and P. Prosser. The distributed asynchronous scheduler. In M. Zweben and M. S. Fox, editors, *Intelligent Scheduling*, pages 309–339. Morgan Kaurmann Publishers, 1994.

[11] M. Davis, G. Logeman, and D. Loveland. A machine program for theorem proving. *Communication of ACM*, 5:394–397, 1962.

[12] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. *Artificial Intelligence*, 68:211–242, 1994.

[13] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[14] J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second Canadian Conference on Artificial Intelligence*, pages 268–277, Toronto, Ont., 1978.

[15] R. Genisson and P. Jegou. Davis and putnam were already forward checking. In *Proceedings of the 12th ECAI*, pages 180–184, 1996.

[16] I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In E. C. Freuder, editor, *Principles and Practice of Constraint Programming*, pages 179–193. Springer, 1996.

[17] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[18] P.van Henteryck, Y. Deville, and C. Teng. A generic arc consistency algorithm and its specializations. In *Artificial Intelligence 57*, pages 291–321, 1992.

[19] A. Mackworth. Consistency in networks of relations. In *Artificial Intelligence 8*, pages 99–118, 1977.

[20] A. K. Mackworth. On reading sketch maps. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 598–606, Cambridge, Mass., 1977.

[21] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, San Jose, Calif., 1992.

[22] R. Mohr and T.C. Henderson. Arc and path consistency revisited. In *Artificial Intelligence 28*, pages 225–233, 1986.

[23] B. A. Nadel. The consistent labeling problem and its algorithms: Towards exact-case complexities and theory-based heuristics, 1986.

[24] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[25] P. Prosser. MAC-CBJ: Maintaining arc consistency with conflict-directed backjumping. Research Report 177, University of Strathclyde, 1995.

[26] J. Règin. A filtering algorithm for constraints of difference in csps. 1994.

[27] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. Technical Report ACT-AI-222-89, MCC, Austin, Texas, 1989. A shorter version appears in *ECAI-90*, pages 550-556.

[28] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 125–129, Amsterdam, 1994.

[29] T. Schiex and G. Verfaillie. Valued constraint satisfaction problems: hard and easy problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 631–639, Montreal, 1995.

[30] Thomas Schiex, Jean-Charles Régin, Christine Gaspin, and Gérard Verfaillie. Lazy Arc Consistency. In *Proceedings of AAAI96*, pages 216–221, Portland, Oregon, USA, 1996.

[31] B. Smith, K. Stergiou, and T. Walsh. Modelling the golomb ruler problem, 1999.

[32] Barbara M. Smith. The brlaz heuristic and optimal static orderings.

[33] Barbara M. Smith and Stuart A. Grant. Trying harder to fail first. In *European Conference on Artificial Intelligence*, pages 249–253, 1998.

[34] Toby Walsh. Sat vs csp. In *Proceedings of CP-2000*, pages 441–456, 2000.