Wilfried Hofstetter & Erica Gralla
16.413

A Tutorial with Java API and Examples on Valued Constraint Satisfaction Problems

1. Introduction

This document is a tutorial on the formulation and solution of 'Valued Constraint Satisfaction Problems'. Valued constraint satisfaction problems are an extension of standard 'Constraint Satisfaction Problems', or CSP's. A standard CSP consists of a series of variables and sets of possible assignments to these variables (domains), along with a set of constraints that relate the allowable assignments among the variables. For example, a crossword puzzle challenges us to fill in blank spaces with words. Crossword puzzles can be seen as CSP's by treating sets of blank spaces as variables, and the word choices as domain values. The constraints arise from the requirement that if two words 'cross', the filled-in letter must be the same in both words.

The constraints in the crossword puzzle are termed 'hard constraints', because they describe conditions that *must* be satisfied in order to achieve a valid solution. It is also possible to envision conditions that we would like to satisfy *as much as possible*. Perhaps a crossword puzzle designer is working for the Walt Disney Company to publicize a re-release of *The Lion King*. Therefore, he wants most of his words to be characters and quotes from the movie. However, if he required all of his words to describe lions and their adventures, he might be unable to create a valid puzzle at all. Therefore, he could create a 'soft constraint' that words should relate to the movie *if possible*. Soft constraints utilize a metric to measure the extent to which the constraint is satisfied; in this case, the metric would measure the number of *Lion King*-related words.

Generally, valued constraint satisfaction problems include both hard and soft constraints; the extreme cases are problems which contain only hard constraints (yielding a standard CSP) and only soft constraints (yielding a pure constraint optimization problem). The established methods for solving these two types of problems are quite different. Constraint optimization problems can be solved using many of the standard optimization techniques, such as gradient search or heuristic methods (e.g. genetic algorithms). Constraint satisfaction problems, on the other hand, can be solved using backtrack search, constraint propagation, and combinations thereof (such as backtrack search with forward-checking). To solve valued CSP's we need to combine these two sets of methods. We can leverage the hard constraints by pruning (deleting) solutions that violate these constraints; this reduces the space of possible solutions. At that point, the solution must be chosen by an optimization technique. There are multiple ways to combine these techniques, including solving a series of branch-and-bound searches with a gradually increasing lower-bound on the soft constraint's metric (e.g. Russian Doll); bucket and mini-bucket elimination algorithms, and combinations of these techniques (the details are not included here; please refer to [1]).

The solution algorithm described and implemented in this tutorial is branch-and-bound search, pruned by a heuristic function called 'mini-bucket heuristics' [1]. It combines the strategies of search and inference. The branch-and-bound/mini-bucket algorithm (BBMB) was chosen because it provides superior performance (in terms of space and time complexity and accuracy of the bounding heuristic function) than other algorithms such as bucket elimination or pure branch-and-bound search [1]. The basic idea of BBMB is to combine a branch-and-bound search (which leverages the hard constraints to prune infeasible solutions) with a mini-bucket heuristic function, which provides an efficient method for pruning sub-optimal solutions. (The details of the mini-bucket heuristic are provided in Chapter 3 of this tutorial.)

The tutorial is organized in the following way: this introduction is followed by a problem description which includes motivation, inputs and outputs, examples, and a description of the Java Application Programmer Interface which contains a generic implementation of the BBMB algorithm. The next chapter provides a description of the algorithm itself and a worked example. Chapter 4 provides the commented source code for the Java API, and Chapter 5 contains a demonstration of the API.

2. Problem Description

2.1 Motivation for Problem

Many real-world resource allocation problems involve both hard and soft constraints. Hard constraints are constraints that always have to be satisfied, whereas soft constraints should be satisfied to the degree that is optimal for the overall system (this optimality is usually captured with a metric function). An example problem from the artificial intelligence domain involving both hard and soft constraints would be path planning with a minimum number of actions (metric function, or soft constraint) given initial and goal states (hard constraints) [1]. Myriad such problems exist in other fields as well, including biology, operations research, engineering design, and business operations. For example, protein structure/folding problems, task or resource scheduling, supply chain management, and design of complex systems each can involve a series of hard and soft constraints. (Note that in some cases other methods also exist for solving these types of problems.)

In addition to problems naturally including both hard and soft constraints, standard CSPs containing only hard constraints can be formulated as constraint optimization problems with minimization of the number of non-satisfied constraints (thus a “best” solution for an unsolvable CSP could be found). This makes all deterministic constraint processing problems tractable with valued CSP solution methods.

The BBMB algorithm implemented here is based on both search and inference. It uses a depth-first-search branch-and-bound algorithm with forward-checking for satisfaction of the hard constraints. The heuristic function employed for a conservative estimate (upper bound for maximization) of the metric function is based on the ‘mini-bucket’ procedure described in more detail in Chapter 3.

2.2 Problem Formulation

This section shows the general problem formulation for valued constraint satisfaction problems alongside the formulation of an example problem, which will be referenced throughout this document.

2.2.1 Example Problem: Antarctic Bases

The example problem concerns a series of remote bases in the Antarctic. A radio network must be set up so that these bases can communicate with the other bases, and with scientists who venture out into the field. Because it is difficult to ship heavy equipment to such remote locations, the bases have only very basic radio equipment. Each base can receive any FM frequency, but can transmit at only one frequency; in addition, weather-ready radio transmitters are only available for a small number of frequencies. Power for the transmitters is also a limited resource: a fixed amount of power is available for sending radio transmissions each day (this is the same across all bases). Each available frequency has a different maximum range for a given power level; therefore, some bases can transmit farther than others. Furthermore, for safety reasons (and for the purposes of scientific collaboration), certain bases *must* be able to communicate with certain other bases, and logistics hubs must transmit at certain frequencies so that they can communicate outside Antarctica. To maximize the amount of area that the scientists can explore, we seek to maximize the total area to which all bases can transmit.

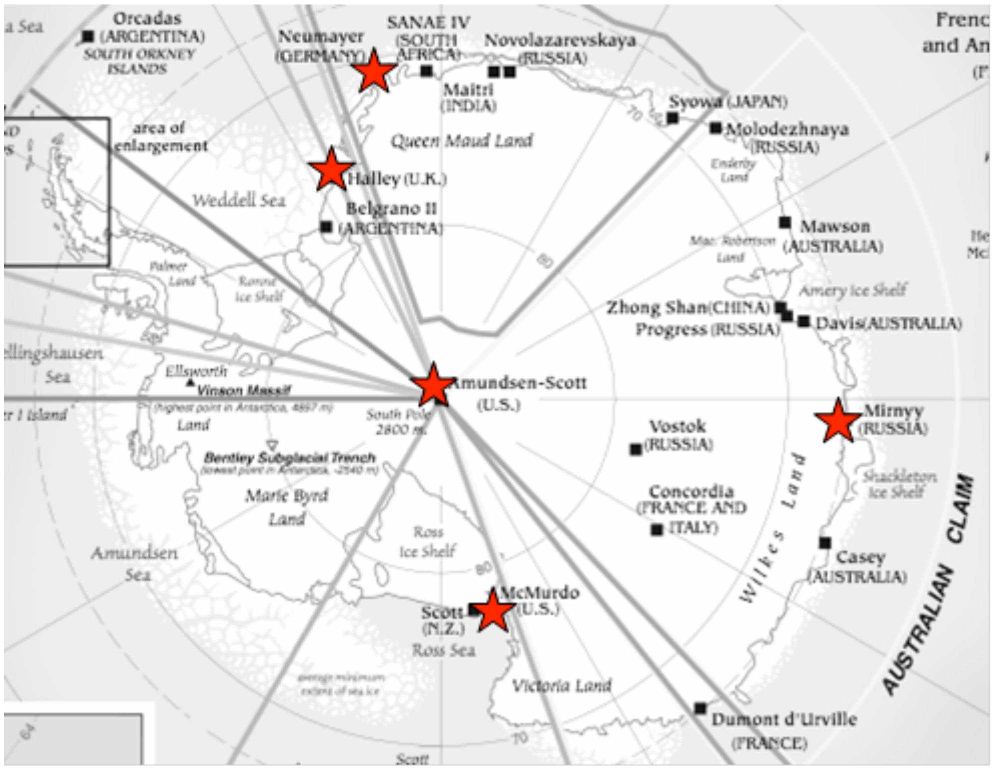

Figure 1: Antarctic bases (bases with red star are used in the example problem)

In one possible scenario, we limit our network to a set of five bases distributed across the Antarctic continent (see map): McMurdo (M), Amundsen-Scott (A), Halley (H), Neumayer (N), and Mirnyy (Y). The available transmitters are at the frequencies 101, 103, and 105 MHz. Lower frequencies can transmit farther for the same power input. McMurdo base is the United States logistics hub, and therefore must be able to communicate with U.S. stations outside the Antarctic, which can only receive at 101 MHz. In addition, every base must be able to communicate with its nearest neighbor.

The goal is to assign radio frequencies to these five bases such that all requirements (hard constraints) are satisfied; in other words, all nearest neighbors can communicate, McMurdo can talk to the 'North', etc.. In addition, we seek to maximize the explorable area of Antarctica by maximizing the total range of all transmitters across all bases.

In the next sections, we present the framework for formulating general problems alongside the formulation for this example.

2.2.2 Inputs and Outputs

The inputs to a constraint optimization problem consist of variables, domains, hard constraints, and soft constraints. The input thus can be written as the 4-tuple

(X, D, C_h, C_s)

These inputs and other important parameters are defined as follows:

- $X = \{ x_1, ..., x_n \}$ are the variables.
- $D = \{ d_1, ..., d_n \}$ are the domains of the variables, and $d_i = \{ a_{i,1}, a_{i,2}, ... \}$ is the set of allowed assignments to variable x_i.
- $\tilde{a} = \{ a_1, ..., a_n \}$ is a set of valid assignments to the variables in X, and $\tilde{a}_i = \{ a_1, ..., a_i \}$ is a *partial assignment:* a set of valid assignments to the first i variables in X.
- C_h is a set of hard constraints (which must be satisfied in order to find a valid solution). Hard constraints are defined in terms of a scope (a set of variables) and a required relationship among the variables. (The example formulation, below, will make this clearer).
- $C_s = \{ F_1, ..., F_l \}$ are the soft constraints, represented as a set of functions each defined over a subset of the variables in X (the scope of the constraint). Each function computes a value, or *metric;* the metrics found by each function are summed to generate the overall metric value for a given set of assignments.

The goal is to maximize the global metric function, which is the sum of all functions in C_s, subject to the constraints C_h. Thus, the solution consists of a value for the global metric function and the set of variable assignments $\tilde{a}^o$ that generates this value.

- $F(\tilde{a}^o) = \max_{\tilde{a}} F(\tilde{a})$, where the global cost function is $F(\tilde{a}) = \sum_{j=1}^{l} F_j(\tilde{a})$.

For the example problem, the variables are the various Antarctic bases, and their domains are the allowed transmission frequencies for each base. In most cases, the domain will contain the whole set of available weather-ready transmitter frequencies, but as stated above, some bases will be restricted only to frequencies at which receivers outside the Antarctic can communicate. This leads to the formulation

- $X = \{ \text{M, A, H, N, Y} \}$
- $D_1 = \{ 101 \}; D_{2\text{-}8} = \{ 101, 103, 105 \}$

The hard constraints state that every base must be able to communicate with its nearest neighbor. In addition, the two U.S. bases must be able to communicate. Thus, the hard constraints link pairs of bases that must be able to talk to each other (the distances between bases are given in section 3.2). These pairs of bases form the *scope* of the constraint; we must now define the required relationship between the variables in the scope. In this case, the hard constraint is satisfied if and only if the two bases can talk to each other; in other words, if the two bases are (1) transmitting on different frequencies and (2) the transmission range of the first base is sufficient to reach the second base. Based on a given set of assignments to the variables, we can determine whether the constraint is violated using this relationship, which we sometimes term a 'consistency check'. Note that these binary hard constraints are *ordered sets*, so it is necessary to include both (M,A) and (A,M).

- C_h = { (M,A), (A,M), (M,Y), (Y,M), (H,N), (N,H) }
- Consistency check:
    - *True* iff $[a_1 \,!= a_2]$ AND $[\text{range}(a_1) \geq \text{distance}(a_1, a_2)]$ ***Equation 1***

The soft constraints give the transmission range of each frequency for a given power level (recall that we seek to maximize this transmission range). We must also capture shrinking of the transmission range due to interference between nearby bases transmitting at the same frequency. We will assume for the purposes of this example that bases continually transmit. Therefore, if two nearby bases transmit on the same frequencies, the two transmissions will interfere anywhere that they overlap, and communication will be impossible in this overlapping region.

In order to simplify the calculations in this example, we will use the *linear* overlap between frequencies rather than calculating the area of overlap (this will allow readers to follow the calculations without complex sector-area formulas). Thus, the soft constraint (below) has a scope that includes all the variables (bases), and a function that captures the linear transmission range of each base. The first term gives the transmission range of each frequency, and the second subtracts the overlapping range if two nearby stations transmit at the same frequency. This equation is the only soft constraint in this example problem.

- $C_s = \{ \, (F_1 = \sum_{i=1}^{n} \left[range(a_i) - (a_i = a_j) \sum_{j=1}^{n} \tfrac{1}{2} overlap(a_i, a_j) \right]) \, \}$ ***Equation 2***

Thus, the global cost function sums the total reach of transmitters for all base stations in Antarctica, accounting for interference. We seek to maximize this quantity. The outputs, therefore, will be the maximum possible reach of transmitters, and the set of assignments of frequencies to bases that generates this situation. A correct solution must include legal assignments to all variables (frequencies to bases), and provide the maximum possible total reach of Antarctic transmitters, while satisfying all the hard constraints.

At this point, we have discussed how to set up and formulate valued constraint satisfaction problems. Before moving on to describe solution methods for these problems, we give a brief overview of the Java-based interface provided for solving these types of problems.

2.3 Java Application Programmer Interface (API)

The Java API for solving valued constraint satisfaction problems consists of five classes:

- ValuedCSPSolver
- Variable
- Constraint
    - HardConstraint
    - SoftConstraint

Each of these classes supplies specific methods for defining, initializing, or solving valued CSPs. They key to this API is that it is not specific to any particular problem formulation: all the methods included are applicable to generic variables sets, domain values, hard constraints, and soft constraints as long as these are consistent with the definition of a valued constraint satisfaction problem. This means that in order to solve an actual problem, the user is required to provide problem-specific classes which extend the generic classes in the API. For example, the generic class ‘HardConstraint’ has a method that checks whether the constraint is consistent. This must be implemented in a problem-specific subclass; in the Antarctic bases problem, the consistency check would return true if the constrained bases can communicate, and false if they cannot.

The following section provides a high-level description of the generic classes and their methods. In this section, we discuss only the *public* methods and classes that are necessary to run the solver. We discuss the internal methods only after the algorithm is described in section 3. The source code for the classes is shown with skeleton method definitions only. Detailed and commented source code for all classes is provided in Chapter 4 of this tutorial. Section 2.4 describes the basics for generating and solving a valued CSP using this API.

Note that the classes in this API do not contain any main methods; in order to solve a problem, a separate class needs to be created that contains a main method.

2.3.1 Class 'ValuedCSPSolver'

This class contains the primary methods for solving a valued CSP. A constructor is available to create a ValuedCSPSolver; the problem can then be initialized using the method "initialize()". The variables, domains, and constraints of the CSP are thus instantiated in the ValuedCSPSolver object. A subclass that extends ValuedCSPSolver (such as AntarcticValuedCSPSolver) must be created so that a problem-specific initialize() method can be written to create the appropriate variables, domains, and constraints.

After the initialization, the method "solveByBnbMiniBucket()" can be run in order to solve the valued CSP defined by the variables and constraints. The methods printProblem() and printResults() send text to the screen describing the problem formulation (variables and constraints) and the solution (value and assignments), respectively. The method setPrintSolutionProgress() is used to determine whether the steps of the algorithm are output to the screen. All other methods are class internal utilities or output functions.

Class overview:

```
public class ValuedCSPSolver {

        // CONSTRUCTOR

        public ValuedCSPSolver()

        // PRINT METHODS

        public void printProblem()
        public void printResults()
        public long getSolutionTime()

        // SOLUTION ALGORITHM

        public double solveByBnbMiniBucket()
        public void initialize()

        // UTILITIES

        public void setPrintSolutionProgress(boolean)
        public void setInitialConst(double)
}
```

2.3.2 Class 'Variable'

The class Variable is used to instantiate variable objects in the ValuedCSPSolver class; this is possible using two different constructors: one specifying only the variable name, and the other specifying the variable name and domain.

The class Variable provides methods to assign domain values to the variable and save the current incumbent variable assignment. It also provides methods for manipulations of domains (pruning, reinstating, etc.) that occur during the branch and bound search. Finally, the class provides utility methods for output and class-internal operations. Please refer to the full source code for more detailed descriptions of each method; extensive comments are included.

Note that this class is independent of the problem formulation; it does not need to be extended for solving a specific problem.

Class Overview:

```
public class Variable {

        // CONSTRUCTORS

        public Variable(String name)
        public Variable(String name, LinkedList domain)

        // ASSIGNMENTS AND INCUMBENTS

        public Object getAssignment()
        public void assign(Object)
        public void assignFirstDomainValue()
        public void clearAssignment()
        public void saveAsIncumbent()

        // DOMAINS

        public void pruneDomain(Object)
        public void pruneDomain(LinkedList)
        public void setDomain(LinkedList)
        public void makeNextLevel()
        public void setNextDomain(LinkedList)
        public LinkedList getDomain()
        public LinkedList get BkpDomains()
        public boolean domainIsEmpty()

        // OTHER

        public void resetVariable(int level)

        // TO-STRING METHODS

        public String toString()
        public String toStringWDomain()
        public String toStringWAssgDomain()
        public String toStringWSolution()
}
```

2.3.3 Class 'Constraint'

The class Constraint is the abstract root class for hard and soft constraints. It provides methods for instantiating a constraint (constructors), for adding associated variables to the constraint, and utility methods.

The class is extended by two specialized classes which are also problem-neutral: HardConstraint and SoftConstraint.

Class Overview:

```
public abstract class Constraint {

        // CONSTRUCTORS

        public Constraint()
        public Constraint(LinkedList variables)

        // VARIABLES

        public void addVariable(Variable)
        public LinkedList getVariables()

}
```

2.3.4 Class HardConstraint

This class extends Constraint and provides specialized methods for creating a hard constraint, for adding variables to a hard constraint, and for checking the consistency of the hard constraint. Of these methods, only the method "checkPairwiseConsistency()" must be implemented by a problem-specific subclass of HardConstraint.

Note that hard constraints are restricted to *two variables* only (binary constraint).

Class Overview:

```
public class HardConstraint extends Constraint {

        // CONSTRUCTORS

        public HardConstraint()
        public HardConstraint(LinkedList variables)
        public HardConstraint(Variable v1, Variable v2)

        // VARIABLES

        public void addVariable(Variable)

        // CONSISTENCY

        public boolean checkConsistency()
        protected boolean checkPairwiseConsistency(Variable v1, Variable v2)
}
```

2.3.5 Class SoftConstraint

This class extends Constraint and provides constructors and a method for evaluating the soft constraint given a partial or a full variable assignment. The evaluation method must be implemented by a problem-specific subclass of SoftConstraint.

Class Overview:

```
public class SoftConstraint extends Constraint {

        public SoftConstraint()
        public SoftConstraint(LinkedList)

        public double evaluate()
}
```

2.3.6 Problem-specific Classes

As described in Sections 2.3.1 – 2.3.5, the following classes need to be extended in problem-specific subclasses in order to define problem-dependent methods:

- ValuedCSPSolver (initialize() method)
- HardConstraint (checkPairwiseConsistency() method)
- SoftConstraint (evaluate() method)

Commented source code examples for extension classes are provided in Chapter 5 of this tutorial.

Please note that the solver assumes a *maximization* problem. In order to perform minimization, simply ensure that the evaluate() method returns the appropriate result multiplied by -1.

2.4 Example: Using the API

After writing the problem-specific subclasses for a given problem, including the initialize() method, a new class must be created with a main method to run the solver. The following code provides a sample 'Tester' class that runs the solver for a problem 'MyVcsp'. (More complete code examples are provided in section 7, and the output from a run of the solver can be found in section 3.3).

Again, please note that the solver assumes a *maximization* problem. In order to perform minimization, simply ensure that the evaluate() method returns the appropriate result multiplied by -1.

```
public class Tester {

    public static void main(String[] args) {

        // create problem-specific solver object
        MyVcspValuedCSPSolver mySolver = new MyVcspValuedCSPSolver();

        // initialize the problem (define variables, domains, and constraints)
        mySolver.initialize();
        // tell the solver whether to print the algorithm steps
        mySolver.setPrintSolutionProgress(true);

        // print the problem definition to the screen
        mySolver.printProblem();

        // run the solution algorithm
        mySolver.solveByBnbMiniBucket();

        // print the results to the screen
        mySolver.printResults();
    }
}
```

3. Method Description: Branch & Bound Mini-Bucket (BBMB)

3.1 Introduction of Big Ideas

3.1.1 Basic Search for Constraint Optimization Problems

Backtrack Search Pruned by Hard Constraints

A straightforward method for solving a constraint optimization problem (X, D, C_h, C_s) is branch-and-bound search based on basic backtrack search: a backtrack search is performed for all variables X and over all domains D in a depth-first order, which essentially examines all possible assignments of domain values to variables. The hard constraints C_h are used to eliminate inconsistent solutions as the backtrack search progresses (this can be accomplished using forward-checking in order to reduce complexity). Once a complete and consistent assignment is found, the total metric function is evaluated based on the soft constraints C_s, and compared to the incumbent U – the "best solution so far". If the new solution is better (i.e. smaller in the case of minimization), it is saved as the new incumbent. The search does not stop when the first complete and consistent assignment is found, because this is not necessarily the global optimum in terms of the metric functions (soft constraints). Search continues until the entire search tree has been explored; at that point, it is clear that the incumbent solution is the optimum.

Backtrack Search Pruned by Hard and Soft Constraints

The method outlined above is a basic backtracking search, augmented only by the use of forward-checking to prune sets of assignments that are inconsistent with the hard constraints. In other words, if a partial assignment generated in the search process is inconsistent, it is pruned, and the search backtracks. However, we can also take advantage of the soft constraints to prune the search tree even further. For each consistent *partial* assignment, the terms of the soft constraints' metric functions can be evaluated and summed into an intermediate metric function. If, in the case of minimization, the value of the intermediate metric function is higher than that of the incumbent, then the partial assignment is pruned and the search backtracks. Thus entire sub-trees of the search tree can be pruned effectively. As before, if a solution is found that has a lower total metric than the incumbent, then the incumbent is updated with the new total cost. The optimum solution is the one which defined the final incumbent.

Adding Heuristic Functions

The basic method described here for pruning based on soft constraints only uses the metric value defined by the *already-instantiated* variables to prune the search tree. However, we could prune more sub-trees if we had some method for estimating the metric value of the remaining *un-instantiated* variables. Such an estimation method, called a *heuristic function*, must be chosen carefully so that branches are not pruned incorrectly due to inaccurate estimates. In minimization problems, it is clear that if a heuristic function never overestimates the metric value from the unassigned variables, we can always prune branches with a higher metric than the incumbent. (For maximization, we require a heuristic that never underestimates). The implementation described above could therefore be viewed as using a heuristic function that is always equal to zero, which is always an admissible heuristic because it never overestimates. However, this heuristic does not provide any additional reduction in complexity. In the following sections we describe heuristics that produce a more accurate lower bound and thus allow for more effective pruning, reducing the complexity of branch-and-bound search.

3.1.2 Branch & Bound Search Using Effective Heuristic Functions

As stated above, an admissible heuristic function must never overestimate the total metric value in a minimization problem. The ideal heuristic function would provide an accurate lower bound (for minimization) of the total metric function given a partial assignment. A straightforward way for achieving this is to use the following relationship:

$$\min\left(C_1(x)+C_2(x)\right)\geq \min C_1(x)+\min C_2(x) \qquad \textbf{\textit{Equation 3}}$$

For given soft constraints / cost functions C_1 and C_2 , the lower bound of the minimum of C_1+C_2 is the sum of the minima of the individual soft constraints (see Figure 2). Thus, the minimum is never overestimated, and the heuristic is admissible. This principle is attractive, because it removes the constraint that the independent variables of C1 and C2 have to have the same values. Because this constraint is removed, it is easier to carry out the minimization.

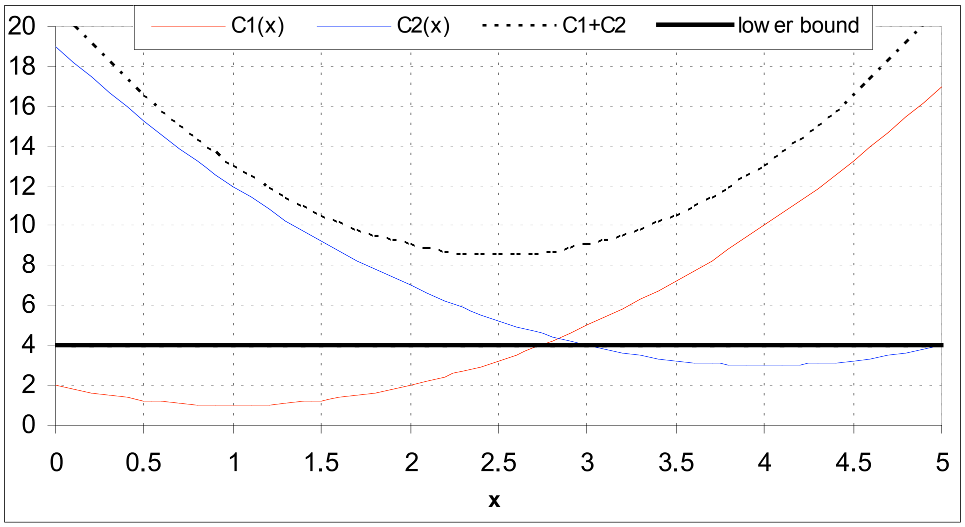

Figure 2: Bounding principle from Equation 3 for two quadratic functions

This principle can be generalized for all soft constraints in a valued CSP, so that given a consistent partial assignment, the lower bound on the minimum total metric can be computed summing the global minima of all the soft constraints. The bounding functions utilizing this principle are called “first-choice” bounding functions [1]. Note that for the computation of the global minima of the individual soft constraints, the hard constraints need to be satisfied.

Another approach for providing an accurate lower bound on the metric function given a partial assignment would be to compute the actual total minimum metric value that is reachable given the current partial assignment. Pruning would then occur if the minimum of the total metric is higher than the incumbent minimum. This could be achieved by using an inference algorithm called “bucket elimination” [1]. For bucket elimination, the individual soft constraints are organized in groups (called “buckets”) according to a variable ordering: all soft constraints mentioning the first variable in the ordering are placed in the first bucket. All remaining soft constraints that include the second variable as independent variable are placed in the second bucket, and so forth. Thus, an optimization operation over only one variable can be performed separately for each bucket; the bucket variable is eliminated by expressing it as a function of the remaining non-initiated variables.

The optimization result is a function of the variables that are not initiated in the bucket (much like when solving, e.g., a duplex integral). The resulting function is placed in the next lower bucket containing one of the non-initiated variables in the function. This top-down process is repeated until the lowest bucket is reached, and a numerical value can be calculated for the last variable. Going through the buckets in bottom-up manner, numerical values for all the other variables can now be calculated.

Note that the variable ordering used for generating the buckets has significant influence on the performance of bucket elimination. Methods for dynamic variable ordering are not described in detail here, but are available in literature for determining optimal variable orderings [1].

3.1.3 Mini-Bucket Elimination as Heuristic Function

Both the first-choice algorithm and bucket elimination have advantages and disadvantages: first-choice produces a fairly inaccurate lower bound, and requires the optimization of all soft constraints for every call of the heuristic function. Bucket elimination provides the actual reachable minimum given a partial assignment of variables, but requires a variable elimination procedure and a subsequent variable calculation procedure (for details of both methods, please refer to [1]).

The mini-bucket elimination algorithm represents an attempt to combine the advantages of both these methods (increased accuracy) while trying to avoid the disadvantages (runtime complexity). Mini-bucket elimination also uses buckets as defined for bucket elimination above. Inside the buckets, it uses the principle from Equation 3 eliminate the identity constraints between the independent variables of two soft constraint functions in a bucket. This avoids the runtime complexity of pure bucket elimination at the cost of reduced accuracy (lower bound instead of actual minimum). Essentially, the mini-bucket elimination heuristic estimates the minimum (or maximum) metric value for the unassigned variables by solving minimization sub-problems over one or several of the remaining variables (rather than all of them at once), thereby trading accuracy for the reduced complexity of smaller, easier minimization problems.

The example presented in the following section walks through the Antarctic bases example using branch-and- bound search with forward checking and a heuristic function based on mini-bucket elimination. Section 3.3 provides the associated pseudocode.

3.1.4 A Note on the Java Implementation of Mini-Bucket Heuristics

This section briefly discusses the Java implementation of the mini-bucket heuristic (which is provided with this tutorial). As noted above, the mini-bucket heuristic is based on the principle that solving a minimization or maximization problem is easier with a smaller number of variables; however, nothing is specified about the choice of the exact number of variables. In this tutorial, we restrict ourselves to mini-buckets containing *only one* variable. While this may not be the optimal choice of mini-bucket size for every problem, it ensures consistency throughout the tutorial, makes examples much easier to follow, and simplifies the code to a more understandable level. It will be straightforward to extend the heuristic function (costToGo() method) so that it can calculate maximums over a larger number of variables.

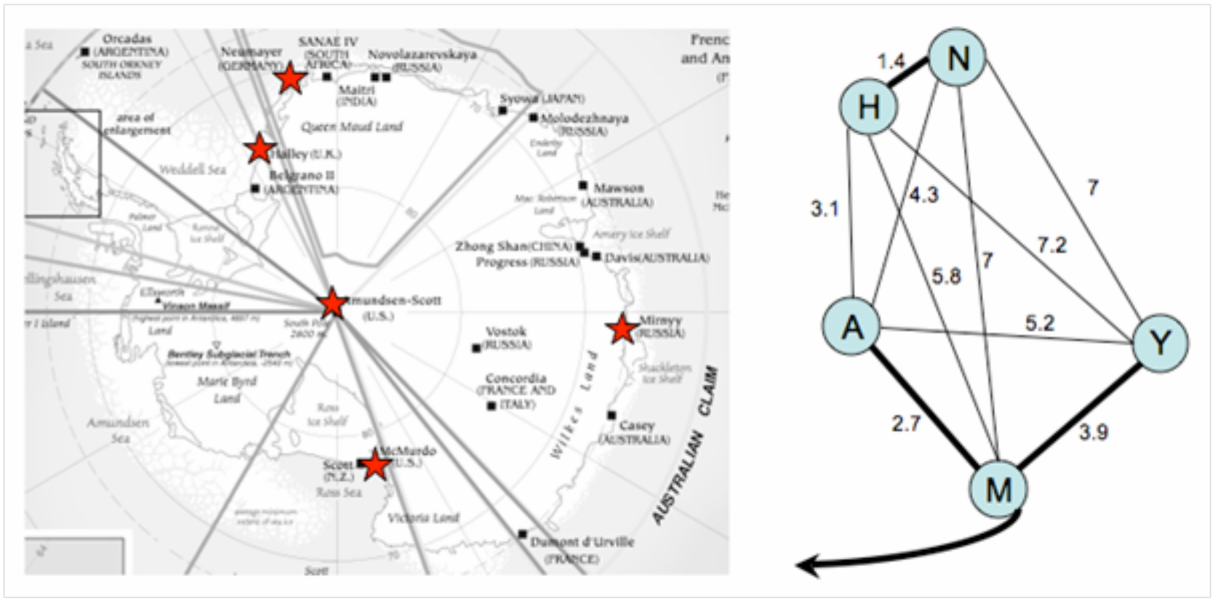

Figure 3: Antarctic Bases Network

3.2 Algorithm Demonstration: Arctic Base Communications

In order to demonstrate the algorithm on a simple example, we utilize the Antarctic base communications problem given in section 2.2.1. Recall that we consider five Antarctic bases: McMurdo (M), Amundsen-Scott (A), Halley (H), Neumayer (N), and Mirnyy (Y). Only three frequencies are available: 101, 103, and 105 MHz. The hard constraints require that McMurdo communicate on 101 MHz (with stations outside Antarctica), and all bases must be able to communicate with their nearest neighbors. The resulting constraints are shown in Figure 2: bold lines indicate that the connected bases must be able to communicate with one another. The distances between bases are shown in the diagram, and the ranges (transmission distances) for each frequency are:

- range(101) = 6
- range(103) = 4
- range(105) = 2

Overlap can be calculated by summing the ranges of two bases and subtracting the distance between them.

Initialize

The inputs to the algorithm are thus:

- An ordered set of variables X = { M, A, H, N, Y } with associated domains D_i
- A set of hard constraints C = { (A,M), (M,Y), (H,N) }
- A soft constraint function F = f(X) = total range – total overlap

Throughout this example, we will represent the search tree as a table, which allows us to show changes in the domains of all variables as the search progresses. Thus, view the table below as the start of a 'sideways' tree, with the current (first) node on the left, with a heavy border. The domains of the variables are printed in the associated boxes in plain text. Assigned values are bolded and underlined, while pruned values are crossed out.

	M	A	H	N	Y
1	101	101,103,105	101,103,105	101,103,105	101,103,105

Assign Value

In order to implement a depth-first search, we loop through all the variables, beginning with the first element in the ordered set X.

x_1 = M

All original variable domains are saved so that they can be restored later. Unless the variable's domain is empty, the variable is assigned a value from its domain (which is then removed from the domain). In our example, we have

$D_{M,t-1}$ = { 101 }
M = 101
$D_{M,t}$ = { }

	M	A	H	N	Y
1	**<u>101</u>**	101,103,105	101,103,105	101,103,105	101,103,105

Forward-Checking

The next step is to *forward-check*, or prune other variables' domain values that are inadmissible due to hard constraints. Thus, all variables connected by hard constraints to the current variable are examined for consistency. In this case, the relevant constraints are those whose scopes include the current variable M:

(A,M), (M,Y)

Bases M and A will not be able to communicate if A is assigned 101 (due to transmission interference), so that value is deleted from the domain of A. The same logic leads the value 101 to be deleted from the domain of Y. Finally, the value 105 is deleted from A and Y, because the range of 105 is not sufficient to reach M in each case. This process eliminates all domain values that are inconsistent with the current assignment based on the hard constraints.

	M	A	H	N	Y
1	**101**	~~101~~,103,~~105~~	101,103,105	101,103,105	~~101~~,103,~~105~~

Metric Heuristic

The next step is to estimate the total metric function (from the soft constraints). The estimated metric value will give the maximum possible metric that the current set of partial assignments can provide. Because this is a maximization problem, if this value is *less than* a previously found solution, there is no need to proceed further down this branch of the search tree. At the moment, there is no previously found, or *incumbent*, solution, but we still estimate the metric for consistency.

First, the metric-so-far (metric based on the current partial assignment) is found from the variables already assigned, by applying Equation 2 to the partial assignment of variables. In this case, we have only one variable, so there is no overlap, and therefore the metric is simply the total transmission range

$f_1 = \text{range}(M) = 6$

Second, the metric-to-go (an estimate of the maximum metric value obtainable from the unassigned variables) is estimated based on the mini-bucket heuristic. It is essential that this process always *overestimate* the true metric-to-go. Therefore, the estimated metric can be calculated by the sum of the maximum possible ranges of all remaining variables (taking into account any overlap with the already assigned variables). This is much simpler (though less accurate) than calculating the maximum range of all remaining assignment sets, which would require maximization over a number of different variables. This method only requires selecting the maximum value from the domain of a single variable. In this case, the variables A, H, N, and Y remain unassigned. Recall that the maximum range is the minimum frequency; however, we must take into account overlap with the current assignments. therefore, even though assigning H a frequency of 101 appears to provide the maximum range, the overlap with transmissions (at 101) from base M reduces the transmission range significantly, so that the choice of 103 actually provides a higher metric value. Again, the metric value is calculated from Equation 2. These values are summed so that

$g_1 = \max(F_1(A))+\max(F_1(H))+\max(F_1(N))+\max(F_1(Y))$

	M	A	H	N	Y
1	**101** f = 6 g = 16	~~101~~,103,~~105~~ max: 4	101,103,105 max: 4	101,103,105 max: 4	~~101~~,*103*,~~105~~ max: 4

The final step of this segment is to check the total cost function f+g against the incumbent solution. However, there is no incumbent solution so we proceed and iterate the whole process with the next variable. The incumbent solution will not be set until the first complete, consistent assignment set is generated; in other words, until the search tree reaches an assignment of the variable 'Y'.

Second Iteration

The second iteration assigns a value (103) to the variable A, and performs forward-checking using the constraint (A,M). This constraint has already been satisfied in the previous step, so no pruning is necessary. The cost-so-far is easily found from the ranges of M and A, and the cost remaining is the sum of the maximum metric values for H, N, and Y. Note that in this case, the maximum value for H comes from the assignment H=105, because assignments of 101 or 103 would each overlap significantly with transmissions from bases M and A. We see this effect in the maximum metric from Y, which must be assigned 103; it overlaps with base A, so its transmission range is reduced from 4 to 1.2.

	M	A	H	N	Y
1	**101** f = 6 g = 16	~~101~~,103,105 max: 4	101,103,105 max: 4	101,103,105 max: 6	~~101~~,103,~~105~~ max: 4
2		**103**, 105 f = 10 g = 6	101,103,105 max: 2	101,103,105 max: 2	103 max: 1.2

Remaining Iterations

The process continues in the same manner. In step 3, the value 101 is pruned from the domain of N, due to the constraint (H,N). The cost-so-far is calculated from the sum of the ranges of M, A, and H, *minus* the overlap between M and H, which are at the same frequency.

The remaining steps can be followed in the output from the code, given in the next section (3.3). The algorithm continues assigning variables until it reaches step 5, which provides a consistent solution with a cost value of 10.3. This then becomes the incumbent. Because there are no remaining values in the domain of Y, the algorithm backtracks to N, restoring the domain of Y to its state *before pruning* at level 4 (the last step in which N was assigned a value).

The value of f+g is calculated to be 13.0, and compared to the incumbent (10.3). If it were less than the incumbent, it would be pruned because it could never generate a solution value higher than that already found. However, since the estimate is greater than the incumbent, the search proceeds normally. With the subsequent assignment of 103 to Y, we find a second consistent solution. The metric value of 13 is higher than the incumbent, so 13 becomes the new incumbent solution.

Search continues in the same manner until the entire search tree is exhausted. In this manner, the true optimal solution is found.

3.3 Complete Algorithm Run: Antarctic Base Communications

This section shows a complete run of the branch-and-bound search with mini-bucket heuristics on the Antarctic base communications example. A basic explanation of the workings of the algorithm is provided below the screen output; please refer to the previous section (3.2) for a more detailed explanation and walk-through of the algorithm. Also, please note that the term "cost" is used in place of "metric" in the code and in the screen output below.

```
Variables:
    M, A, H, N, Y,
Domains:
    (M: 101), (A: 101 103 105), (H: 101 103 105), (N: 101 103 105), (Y: 101 103 105),
Hard Constraints:
    (M A ), (A M ), (M Y ), (Y M ), (H N ), (N H ),
Soft Constraints:
    (M A H N Y ),

Level 0 (before pruning)
    (M: *101*) (A: 101 103 105) (H: 101 103 105) (N: 101 103 105) (Y: 101 103 105)
Level 0 (after pruning)
    (M: *101*) (A: 103) (H: 101 103 105) (N: 101 103 105) (Y: 103)
    Cost So Far: 6.0
    Cost To Go:  16.0

Level 1 (before pruning)
    (M: *101*) (A: *103*) (H: 101 103 105) (N: 101 103 105) (Y: 103)
Level 1 (after pruning)
    (M: *101*) (A: *103*) (H: 101 103 105) (N: 101 103 105) (Y: 103)
    Cost So Far: 10.0
    Cost To Go:  5.199999999999999

Level 2 (before pruning)
    (M: *101*) (A: *103*) (H: *101* 103 105) (N: 101 103 105) (Y: 103)
Level 2 (after pruning)
    (M: *101*) (A: *103*) (H: *101* 103 105) (N: 103 105) (Y: 103)
    Cost So Far: 9.8
    Cost To Go:  3.1999999999999993

Level 3 (before pruning)
    (M: *101*) (A: *103*) (H: *101* 103 105) (N: *103* 105) (Y: 103)
Level 3 (after pruning)
    (M: *101*) (A: *103*) (H: *101* 103 105) (N: *103* 105) (Y: 103)
    Cost So Far: 10.100000000000001
    Cost To Go:  0.1999999999999993

Level 4 (before pruning)
    (M: *101*) (A: *103*) (H: *101* 103 105) (N: *103* 105) (Y: *103*)
Level 4 (after pruning)
    (M: *101*) (A: *103*) (H: *101* 103 105) (N: *103* 105) (Y: *103*)
    Cost So Far: 10.3
    Cost To Go:  0.0
Incumbent assigned: 10.3

Level 3 (before pruning)
    (M: *101*) (A: *103*) (H: *101* 103 105) (N: *105*) (Y: 103)
Level 3 (after pruning)
    (M: *101*) (A: *103*) (H: *101* 103 105) (N: *105*) (Y: 103)
    Cost So Far: 11.8
    Cost To Go:  1.1999999999999993
```

```
Level 4 (before pruning)
   (M: *101*) (A: *103*) (H: *101* 103 105) (N: *105*) (Y: *103*)
Level 4 (after pruning)
   (M: *101*) (A: *103*) (H: *101* 103 105) (N: *105*) (Y: *103*)
   Cost So Far: 13.0
   Cost To Go:  0.0
Incumbent assigned: 13.0

Level 2 (before pruning)
   (M: *101*) (A: *103*) (H: *103* 105) (N: 103 105) (Y: 103)
Level 2 (after pruning)
   (M: *101*) (A: *103*) (H: *103* 105) (N: 105) (Y: 103)
   Cost So Far: 9.100000000000001
   Cost To Go:  2.3999999999999986
Pruned! Est. cost of 11.5 is less than incumbent 13.0

Level 3 (before pruning)
   (M: *101*) (A: *103*) (H: *103* 105) (N: *105*) (Y: 103)
Level 3 (after pruning)
   (M: *101*) (A: *103*) (H: *103* 105) (N: *105*) (Y: 103)
   Cost So Far: 11.100000000000001
   Cost To Go:  0.3999999999999986
Pruned! Est. cost of 11.5 is less than incumbent 13.0

Level 4 (before pruning)
   (M: *101*) (A: *103*) (H: *103* 105) (N: *105*) (Y: *103*)
Level 4 (after pruning)
   (M: *101*) (A: *103*) (H: *103* 105) (N: *105*) (Y: *103*)
   Cost So Far: 11.5
   Cost To Go:  0.0
Pruned! Est. cost of 11.5 is less than incumbent 13.0

Level 2 (before pruning)
   (M: *101*) (A: *103*) (H: *105*) (N: 105) (Y: 103)
Level 2 (after pruning)
   (M: *101*) (A: *103*) (H: *105*) (N:) (Y: 103)
   Cost So Far: 12.0
   Cost To Go:  1.1999999999999993

Search completed! Best solution: 13.0
```

Initially, the problem description is printed on the screen showing problem variables, associated domains, hard constraints and their associated variables, and soft constraints and their associated variables (in this case there is only one soft constraint which is a function of all variables). Then a series of outputs similar in form follows: these are the individual search steps of the branch and bound algorithm. "Level" denotes the layer of the search tree the algorithm is currently at; each level corresponds to one variable and its assignments. Level 0 corresponds to the first variable.

Cost-so-far denotes the metric value (sum of soft constraint function evaluations) calculated based on a partial assignment to variables. Cost-to-go is the value of the bounding (and therefore admissible) heuristic function for the sub-tree of unassigned variables given the current partial variable assignment. In addition to the two cost components of the branch and bound evaluation function, the current consistent domain values for all variables are printed out before and after forward-checking, and the currently assigned domain values are indicated by a star at the start and end. Also shown is the action of pruning a variable assignment when the sum of the cost-so-far and the cost-to-go are lower than the current best solution (the incumbent).

This output block is repeated for each step in the search until the search tree is exhausted (all variable domains exhausted); the search terminates with the best values for the metric function found (which is the case here), or a statement that the search failed otherwise.

3.4 Pseudocode

The process described above can be distilled into general ‘pseudocode’, given below.

```
given
    variables $X = \{ x_1 \dots x_n \}$
    domains $D_x = \{ d_1 \dots d_m \}$
    hard constraints $C = \{ c_1 \dots c_p \}$
    soft constraints $F = \{ f_1 \dots f_q \}$

for all variables $x_i = x_1$ to $x_n$

    *assign value from domain*
    save backup domains for all variables $B^i = \{ D^i \}$
    if domain $D_x^i$ is empty, then backtrack: return to previous variable
    assign next domain value to current variable $x_i = d_{next}$
    remove assigned domain value $d_{next}$ from domain $D_x^i$

    *forward-check: prune by hard constraints*
    for all hard constraints $c_k = c_1$ to $c_p$
        if $c_k$ references $x_i$
            if $c_k$ is not consistent
                remove the inconsistent value from the domain of the referenced variable
            end if
        end if
    end for

    *calculate cost-so-far*
    for all soft constraints $f_l = f_1$ to $f_q$
        $f^i$ += $f_l(x_1 \dots x_i)$
    end for

    *calculate cost-to-go*
    for all soft constraints $f_l = f_1$ to $f_q$
        for all unassigned variables $x_{i'} = x_{i+1}$ to $x_n$
            for all values in the domain $d_j = d_1$ to $d_m$
                if $g_l(d_{i'})$ > previously found $g_l(x_{i'})$, save max( $g_l(x_j)$ ) = $g_l(d_{i'})$
            end for
            $g^i$ += max( $g_l(x_j)$ )
        end for
    end for

    *prune by heuristic*
    if f+g < incumbent, then backtrack: assign new domain value
end for
```

4. API Java Implementation

This chapter is intended as a reference for the implementation of the classes implemented in the Valued-Constraint-Satisfaction API. In order to make this tutorial easier to read, the source code for the API is provided in Appendix 7.1 at the end of the tutorial. Information on problem-specific classes is provided with the examples in Chapter 5 and in Appendices 7.2 and 7.3. The source code is commented richly and should be the primary source of information on how the classes are implemented.

We consolidate here a few notes (already discussed in earlier sections) pertaining to the use of the Java API provided with this tutorial.

In order to use the API to solve a specific valued CSP, several generic subclasses need to be extended and certain methods implemented. As described in Chapter 2, the generic classes are implemented in a problem-neutral way; in order to solve an actual problem, problem-specific versions of certain classes have to be supplied. The methods that must be implemented are:

-
- ValuedCSPSolver (initialize() method)
- HardConstraint (checkPairwiseConsistency() method)
- SoftConstraint (evaluate() method)

Section 2.3 provided a top-level introduction to these classes and the methods provided within.

Also note that the implementation assumes a *maximization* problem. In order to solve a minimization problem, simply return negative values (multiply the metric/cost value by -1).

In order to run the solver, a new class must be written that contains a main method. Examples showing how to write this method are given in section 2.4.

Again, the source code is the best source of information on how the algorithm is implemented. The comments are intended to explain how the algorithm works step-by-step. This code is also reprinted in section 5.

5. API Demonstration

In order to provide maximum flexibility to the user, the implementation of a valued constraint satisfaction problem as described in this tutorial consists of problem-specific classes and generic classes for solving the valued CSP. Section 5.1 describes how to load and use the generic classes, and Sections 5.2 and 5.3 provide examples for problem-specific classes.

5.1 Loading and Using the 'Valued_CSP' Package

As described in Section 2.3, there are five generic classes in the API:

- ValuedCSPSolver
- Variable
- Constraint
    - HardConstraint
    - SoftConstraint

These classes form a package called 'Valued_CSP' (see source code in the Appendix (7)). In order to be able to use the package, it has to be imported for all problem-specific classes using the following command (see source code in Sections 5.2 and 5.3):

```
import Valued_CSP.*;
```

In order to be able to access the files in the package, a folder / subdirectory called 'Valued_CSP' which contains all the classes in the package has to be present in the same directory as the problem-specific source classes.

In order to actually run the solver, a main class or a Junit test class has to be provided, and an object of the problem-specific ValuedCSPSolver class has to be instantiated (see examples below).

5.2 Source Code for Antarctic Communications Example

The Antarctic communications problem has been described in detail in the preceding Chapters. Here we provide a description of the source code for this problem as an example for implementing and solving a particular problem using the Valued CSP API. The actual source code is shown in Appendix 7.2.

The example implementation consists of five user-defined classes:

AntarcticValuedCSPSolver

This class extends ValuedCSPSolver from the API and provides a problem-specific version of the method initialize(). In this method, the variables, their domains, the hard constraints, and the soft constraints are defined and initialized. The variables are the Antarctic bases, their domain values are frequencies stored as String objects (for detailed example description, see Chapter 3 above). Hard constraints are defined between appropriate variables. There is only one soft constraint, which is a function of all the variables.

AntarcticHardConstraint

This class extends HardConstraint from the API provides a problem-specific version of the method checkPairwiseConsistency(). Hard constraints are defined for pairs of bases that need to be able to talk to each other. The method returns true if at least one of the two variables is unassigned. The method returns false if the bases use the same frequency, if their mutual ranges are smaller than the distances between them, and if McMurdo ("M") is not assigned the frequency 101.

In order to carry out the evaluations of these constraints, methods for calculating range and distance are used; these methods are provided by the problem-specific class AntarcticUtilities.

AntarcticSoftConstraint

This class extends SoftConstraint from the API; it provides a problem-specific version of the method evaluate(). The evaluation function returns the sum of all the exploration ranges of the bases minus the overlap between bases.

If a variable (which represents a base) is unassigned, the evaluation function does not calculate a range for it, and does not subtract any overlap.

AntarcticUtilities

This class does not extend any of the API classes; it is an abstract class, purely user-defined and provides methods for calculating the communications range of a base given a frequency and for looking up the distance between two bases. These methods are used by the problem-specific soft and hard constraints.

For many valued constraint satisfaction problems it might be desirable to separate the calculation of numerical data in a separate class in order to simplify the classes for soft and hard constraints and to facilitate changing the data at a later point in time.

AntarcticTester

None of the classes discussed thus far have contained a main method. The class Tester contains a main class and is used to run the Antarctic communications example. In the main class, an object of class AntarcticValuedCSPSolver is defined and initialized. After that, the solution algorithm is called for solving the problem.

The output from running the Antarctic communications example is provided in Section 3.3 above. For more detailed information on the implementation, please refer to the documented source code in Appendix 7.2.

5.3 Source Code and Benchmark Analysis for Uncapacitated Warehouse Location Problem (UWLP)

The Antarctic communications example above is intended to provide a motivating and easily understandable introduction to formulating and solving a real-world valued CSP. However, because it is not easily scalable, another example problem of somewhat more academic nature is provided here: the so-called Uncapacitated Warehouse Location Problem (UWLP).

5.3.1 Problem Description

The UWLP is concerned with determining the optimal location for warehouses given w potential locations and the associated cost (negative revenue) of maintaining a warehouse at that location. In addition, there are s stores which need to be supplied by the warehouses; each store is supplied by exactly one warehouse (whereas one warehouse can supply multiple stores). The added cost (negative revenue) for supplying a store from a warehouse is a function of the warehouse location and the store location.

The UWLP presented here is simplified so that it has only binary constraints between the warehouses and the stores as opposed to n-ary constraints used in the generic UWLP formulation [4]; this formulation was inspired by [3].

The variables for this problem are the warehouses and the stores. For the warehouses, the domains consist of "Y" (the warehouse is operated) and "N" (the warehouse is not operated). For the store variables, the domains consist of the names of the warehouses (independent of their existence). For w warehouses and s stores there are **w + s** variables.

The hard constraints for this problem enforce that in order to be supplied from a particular warehouse, that warehouse must be operated ("Y"). If not, the warehouse name is pruned from the store domain. According to this definition, there are **w * s** hard constraints between the w warehouses and the s stores.

The goal is the minimization of the cost for supplying the s stores from at most w different warehouses. As our algorithm provides a tool for maximization, the goal is reformulated for maximizing the negative cost (revenue). The metric function / soft constraint is therefore the sum of all the negative costs arising from operating warehouses and due to supplying a particular store from these warehouses.

For the problem formulation provided here, only one soft constraint is required.

A short description of the problem specific classes is provided here, while the commented source code is provided in Appendix 7.3.

Class UWLP_ValuedCSPSolver

This class extends ValuedCSPSolver from the API. A problem-specific version of the initialize() method is provided as well as a method for setting the number of warehouses and stores: setNumberWarehousesAndStores(). In the initialize method, the variables, hard, and soft constraints are defined.

Class UWLP_HardConstraint

This class extends HardConstraint from the API. It provides a customized checkPairwiseConsistency() method for the UWLP. The constraint is always initialized with one warehouse variable and one store variable. The constraint returns true if at least one of the two variables is unassigned. It returns false if the warehouse variable is assigned "N" and the store variable is assigned the name of the warehouse variable.

Class UWLP_SoftConstraint

This class extends SoftConstraint from the API and provides a user-specific method evaluate()._As mentioned above, the UWLP problem formulation provided here utilizes only one soft constraint / metric function.

The method evaluate() returns a double value corresponding to the current overall revenue (negative cost). Due to the way the problem was formulated, this value will always be smaller or equal zero.

The method evaluate() only calculates the revenue of a warehouse if the warehouse is operated (i.e. assigned "Y"), and it only calculates the revenue for a store-warehouse combination if both variables are assigned.

Class UWLP_Tester

Again, a class is required that contains a main method for running the problem (alternatively a Junit test class could be used). The class UWLP_Tester provides the main method. In this method, an object of class UWLP_ValuedCSPSolver is created. Then the number of warehouses and stores is specified using the method setNumberWarehousesAndStores(). After that, the problem is initialized using initialize(), and then the solver is run. Finally, the results are printed to the screen using printResults().

Note: the ability to specify the number of warehouses and stores make the problem scalable which is important for benchmarking.

For details of the code and instantiating and solving a problem in the UWLP_Tester method, please refer to the commented source code in Appendix 7.3.

5.3.2 UWLP Example Run

One of the motivating factors behind our use of the UWLP example in this tutorial is that the problem is scalable; in other words, we can use the UWLP_Tester class to specify the number of variables, so the problem can be run for different test cases. As for the Antarctic communications problem, a sample screen output is provided here to show the working of the algorithm. The example run shown here is for 2 warehouses and 2 stores, which is one of the simplest cases for which pruning can occur without pruning the entire variable domain of a store.

The first output is the problem definition showing variables and domains, as well as hard and soft constraints. Note that the warehouse variables "W" have domains {"Y", "N"}, whereas the store variables "S" have domains that consist of the warehouse names. Hard constraints always exist between one warehouse and one store, and the soft constraint includes all stores and warehouses as variables.

After the problem definition, the algorithm starts going through the tree. If a warehouse is assigned "N", then the warehouse name is pruned from all store domains during the forward checking step. The cost-so-far and cost-to-go functions are always smaller or equal 0. Below the final solution, the solution time in milliseconds is provided for benchmarking purposes.

Screen output:

```
PROBLEM DEFINITION
Variables:
   W0, W1, S0, S1,
Domains:
   (W0: Y N), (W1: Y N), (S0: W0 W1), (S1: W0 W1),
Hard Constraints:
   (W0 S0 ), (W1 S0 ), (W0 S1 ), (W1 S1 ),
Soft Constraints:
   (W0 W1 S0 S1 ),

Level 0 (before pruning)
   (W0: *Y* N) (W1: Y N) (S0: W0 W1) (S1: W0 W1)
Level 0 (after pruning)
   (W0: *Y* N) (W1: Y N) (S0: W0 W1) (S1: W0 W1)
   Cost So Far: -10.0
   Cost To Go:  -40.0

Level 1 (before pruning)
   (W0: *Y* N) (W1: *Y* N) (S0: W0 W1) (S1: W0 W1)
Level 1 (after pruning)
   (W0: *Y* N) (W1: *Y* N) (S0: W0 W1) (S1: W0 W1)
   Cost So Far: -22.0
   Cost To Go:  -40.0

Level 2 (before pruning)
   (W0: *Y* N) (W1: *Y* N) (S0: *W0* W1) (S1: W0 W1)
Level 2 (after pruning)
   (W0: *Y* N) (W1: *Y* N) (S0: *W0* W1) (S1: W0 W1)
   Cost So Far: -41.0
   Cost To Go:  -21.0

Level 3 (before pruning)
   (W0: *Y* N) (W1: *Y* N) (S0: *W0* W1) (S1: *W0* W1)
Level 3 (after pruning)
   (W0: *Y* N) (W1: *Y* N) (S0: *W0* W1) (S1: *W0* W1)
   Cost So Far: -62.0
```

```
    Cost To Go:  0.0
Incumbent assigned: -62.0

Level 3 (before pruning)
    (W0: *Y* N) (W1: *Y* N) (S0: *W0* W1) (S1: *W1*)
Level 3 (after pruning)
    (W0: *Y* N) (W1: *Y* N) (S0: *W0* W1) (S1: *W1*)
    Cost So Far: -64.0
    Cost To Go:  0.0
Pruned! Est. cost of -64.0 is less than incumbent -62.0

Level 2 (before pruning)
    (W0: *Y* N) (W1: *Y* N) (S0: *W1*) (S1: W0 W1)
Level 2 (after pruning)
    (W0: *Y* N) (W1: *Y* N) (S0: *W1*) (S1: W0 W1)
    Cost So Far: -43.0
    Cost To Go:  -21.0
Pruned! Est. cost of -64.0 is less than incumbent -62.0

Level 3 (before pruning)
    (W0: *Y* N) (W1: *Y* N) (S0: *W1*) (S1: *W0* W1)
Level 3 (after pruning)
    (W0: *Y* N) (W1: *Y* N) (S0: *W1*) (S1: *W0* W1)
    Cost So Far: -64.0
    Cost To Go:  0.0
Pruned! Est. cost of -64.0 is less than incumbent -62.0

Level 3 (before pruning)
    (W0: *Y* N) (W1: *Y* N) (S0: *W1*) (S1: *W1*)
Level 3 (after pruning)
    (W0: *Y* N) (W1: *Y* N) (S0: *W1*) (S1: *W1*)
    Cost So Far: -66.0
    Cost To Go:  0.0
Pruned! Est. cost of -66.0 is less than incumbent -62.0

Level 1 (before pruning)
    (W0: *Y* N) (W1: *N*) (S0: W0 W1) (S1: W0 W1)
Level 1 (after pruning)
    (W0: *Y* N) (W1: *N*) (S0: W0) (S1: W0)
    Cost So Far: -10.0
    Cost To Go:  -40.0

Level 2 (before pruning)
    (W0: *Y* N) (W1: *N*) (S0: *W0*) (S1: W0)
Level 2 (after pruning)
    (W0: *Y* N) (W1: *N*) (S0: *W0*) (S1: W0)
    Cost So Far: -29.0
    Cost To Go:  -21.0

Level 3 (before pruning)
    (W0: *Y* N) (W1: *N*) (S0: *W0*) (S1: *W0*)
Level 3 (after pruning)
    (W0: *Y* N) (W1: *N*) (S0: *W0*) (S1: *W0*)
    Cost So Far: -50.0
    Cost To Go:  0.0
Incumbent assigned: -50.0
```

```
Level 0 (before pruning)
    (W0: *N*) (W1: Y N) (S0: W0 W1) (S1: W0 W1)
Level 0 (after pruning)
    (W0: *N*) (W1: Y N) (S0: W1) (S1: W1)
    Cost So Far: 0.0
    Cost To Go:  -44.0

Level 1 (before pruning)
    (W0: *N*) (W1: *Y* N) (S0: W1) (S1: W1)
Level 1 (after pruning)
    (W0: *N*) (W1: *Y* N) (S0: W1) (S1: W1)
    Cost So Far: -12.0
    Cost To Go:  -44.0
Pruned! Est. cost of -56.0 is less than incumbent -50.0

Level 2 (before pruning)
    (W0: *N*) (W1: *Y* N) (S0: *W1*) (S1: W1)
Level 2 (after pruning)
    (W0: *N*) (W1: *Y* N) (S0: *W1*) (S1: W1)
    Cost So Far: -33.0
    Cost To Go:  -23.0
Pruned! Est. cost of -56.0 is less than incumbent -50.0

Level 3 (before pruning)
    (W0: *N*) (W1: *Y* N) (S0: *W1*) (S1: *W1*)
Level 3 (after pruning)
    (W0: *N*) (W1: *Y* N) (S0: *W1*) (S1: *W1*)
    Cost So Far: -56.0
    Cost To Go:  0.0
Pruned! Est. cost of -56.0 is less than incumbent -50.0

Level 1 (before pruning)
    (W0: *N*) (W1: *N*) (S0: W1) (S1: W1)
Level 1 (after pruning)
    (W0: *N*) (W1: *N*) (S0:) (S1:)
    Cost So Far: 0.0
    Cost To Go:  -200000.0
Pruned! Est. cost of -200000.0 is less than incumbent -50.0

Search completed! Best solution: -50.0
62
```

5.3.3 Benchmarking

For purposes of comparing the performance of the algorithm implemented in the API with optimized ("professional") algorithms, the UWLP problem was run for varying numbers of variables (warehouses and stores) from 6 to 13. Variable numbers over 13 resulted in running times over five minutes, and results are not shown here. Figure 4 provides an overview of the results of the benchmarking. Results from the API algorithm are shown in blue; the black lines represent results using purely inference-based algorithms shown in [3] (complete tight case, page 5). The performance curve clearly grows exponentially with the number of variables.

It can be seen that the inference-based algorithm is about a factor 30 faster than our branch and bound algorithm using mini-bucket heuristics. For details of the inference-based algorithms, please refer to [3]. This is somewhat reasonable given that the algorithm and code presented here was intended to teach the algorithm, rather than solve problems extremely rapidly. The code could most likely be further optimized.

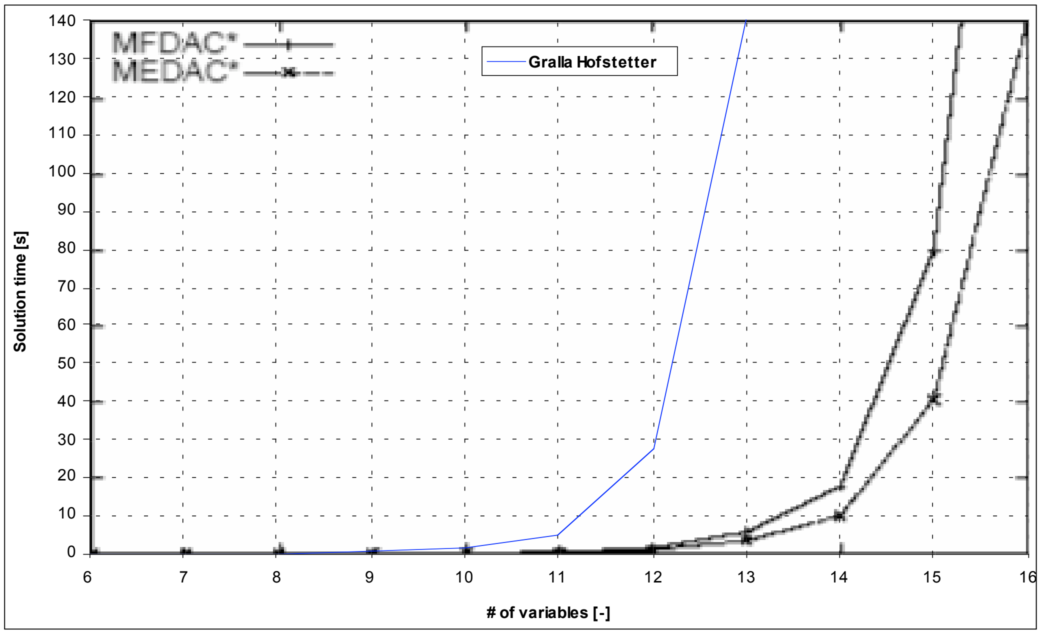

Figure 4: Benchmarking results

6. References

[1] Dechter, R., ***Constraint Processing***, Chapter 13, San Francisco: Morgan Kaufmann Publishers, 2003.
[2] Sachenbacher, M. , Williams, B., ***On-Demand Bound Computation for Best-First Constraint Optimization***, CSAIL, MIT 2004
[3] de Givry, S.; et al, ***Existential arc consistency: Getting closer to full arc consistency in weighted CSPs***_INRA, Toulouse, 2002
[4] Kratica, J., et al, ***Solving the Uncapacitated Warehouse Location Problem By SGA with Add-Heuristic***, Faculty of Mechanical Engineering, Department of Mathematics, Belgrade

-