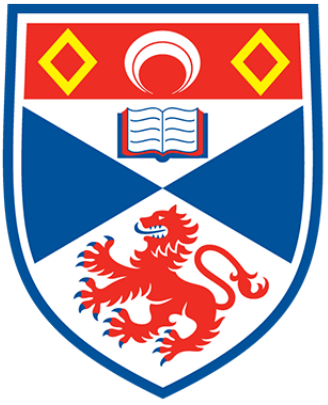


University of
St Andrews

CS4402 Constraint Programming

Revision Notes

May 20, 2018

Lecturer:
Ian Miguel

Submitted By:
140011146

Contents

1 Introduction **2**
- 1.1 Constraint satisfaction problem . . . 2
  - 1.1.1 Constraint modelling . . . 3
  - 1.1.2 Constraint solving . . . 3
- 1.2 Constraint optimisation problem . . . 3
- 1.3 Problem classes . . . 4
- 1.4 Constraint representation . . . 4
  - 1.4.1 Table constraint . . . 4
  - 1.4.2 Intensional constraints . . . 4
- 1.5 Constraint languages . . . 5
- 1.6 Crystal maze example . . . 5

2 Modelling **6**
- 2.1 Abstract constraint models . . . 6
- 2.2 Sequences . . . 7
  - 2.2.1 Fixed-length sequences . . . 7
  - 2.2.2 Bounded-length sequences . . . 8
  - 2.2.3 Unbounded sequences . . . 9
  - 2.2.4 Permutations . . . 9
  - 2.2.5 Viewpoints . . . 10
- 2.3 Sets . . . 10
  - 2.3.1 Fixed-cardinality sets . . . 10
  - 2.3.2 Explicit representation . . . 10
  - 2.3.3 Occurrence representation . . . 11
  - 2.3.4 Set constraints . . . 12
  - 2.3.5 Set intersection . . . 12
  - 2.3.6 Set union . . . 13
  - 2.3.7 Set subset . . . 13
- 2.4 Multisets . . . 13
  - 2.4.1 Explicit representation . . . 14
  - 2.4.2 Occurrence representation . . . 14
- 2.5 Relations . . . 14
  - 2.5.1 Projection . . . 14
  - 2.5.2 Other representations . . . 14
- 2.6 Functions . . . 14
  - 2.6.1 Injections . . . 15
- 2.7 Nested structures . . . 15
  - 2.7.1 The Gripper Problem . . . 15
  - 2.7.2 Nesting inside sequences . . . 16
  - 2.7.3 Nesting inside sets . . . 16
  - 2.7.4 Relations as sets of tuples . . . 17
- 2.8 Symmetry . . . 17
  - 2.8.1 Variable symmetry . . . 17
  - 2.8.2 Value symmetry . . . 18
  - 2.8.3 Consequences of symmetry . . . 18
  - 2.8.4 Symmetry-breaking constraints . . . 19
  - 2.8.5 Lex-leader method . . . 19
  - 2.8.6 Lex^2 ordering . . . 20
- 2.9 Viewpoints . . . 20
  - 2.9.1 Auxiliary variables . . . 20
- 2.10 Branching variables . . . 21
- 2.11 Implied constraints . . . 21

3 Search **22**

3.1 Definitions . . . 22
3.1.1 Systematic search . . . 22
3.1.2 Tree traversal . . . 22
3.1.3 Ordering . . . 23
3.2 Methods . . . 23
3.2.1 Generate and test . . . 23
3.2.2 The backtrack algorithm . . . 24
3.2.3 Branch and bound . . . 24
3.3 Graphical representation . . . 25
3.3.1 Binary constraint graph . . . 25
3.3.2 Dual representation . . . 26

4 Propagation **26**
4.1 Consistency properties . . . 26
4.1.1 Local node consistency . . . 27
4.1.2 Global node consistency . . . 27
4.1.3 Local arc consistency . . . 27
4.1.4 Global arc consistency . . . 27
4.2 Enforcing arc consistency . . . 28
4.2.1 AC1 . . . 28
4.2.2 AC3 . . . 28
4.2.3 AC4 . . . 29
4.2.4 AC2001/3.1 . . . 30
4.3 Forward checking . . . 30
4.4 Heuristics . . . 31
4.4.1 Static variable order heuristics . . . 31
4.4.2 Dynamic variable order heuristics . . . 32
4.4.3 Value assignment order . . . 32
4.4.4 Variable order . . . 33
4.4.5 Value order . . . 33
4.5 Maintaining arc consistency . . . 33

1 Introduction

1.1 Constraint satisfaction problem

The constraint satisfaction problem is a general way in which decision-making problems can be represented and solved. It is defined as follows: **Given**

- A set of **decision variables** - A decision variable corresponds to a choice that must be made in solving a problem. For example in university timetabling, various things have to be decided, such as time for each lecture and venue for each lecture
- For each decision variable, a **domain** of potential values - Values in the domain of a decision variable correspond to the options for a particular choice. When a decision variable is assigned a single value from its domain, it is equivalent to the choice associated with that variable being made.
- A set of **constraints** on the decision variables. The **scope** of a constraint is a subset of the decision variables a constraint involves. Out of all possible combinations of assignments to the variables in its scope, a constraint specifies which assignments are allowed (satisfy the constraint) and which are disallowed (violet the constraint).

Find an **assignment** of values to variables such that all constraints are satisfied.

There are two main phases to solving problems with constraints: **Modelling** and **Solving**.

1.1.1 Constraint modelling

A constraint model maps the features of a given problem onto the features of a constraint satisfaction problem. In other words, the input problem must be modelled in a way that is represented with decisions variables, domains and constraints.

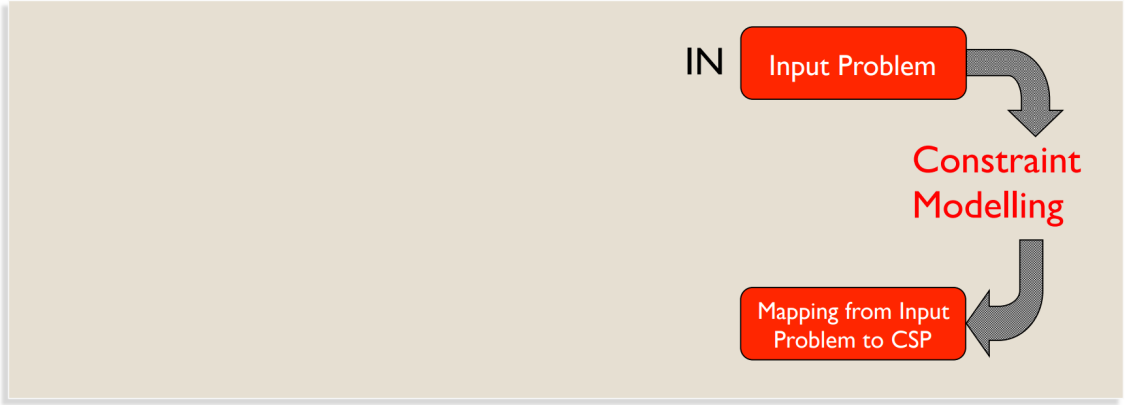


Figure 1: Constraint modelling is the first step in solving constraint problems.

1.1.2 Constraint solving

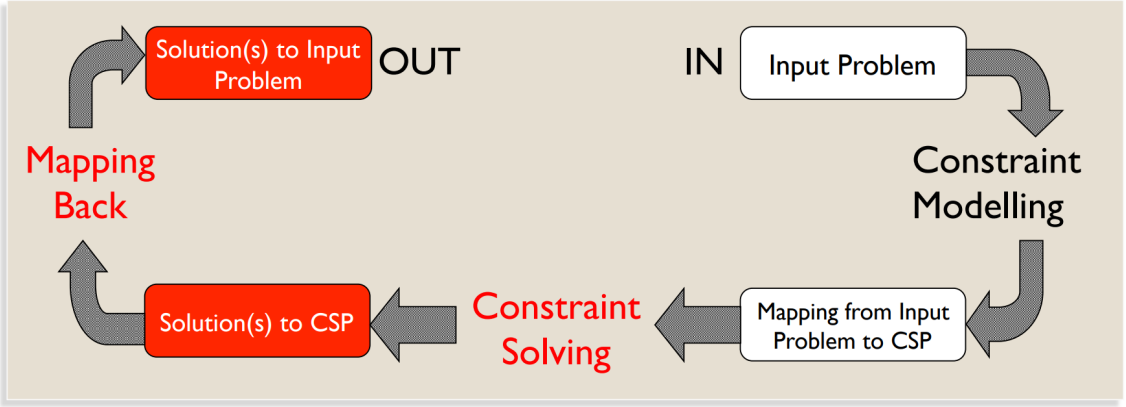


Figure 2: Full pipeline of constraint solving.

The modelled CSP is used as input to a constraint solver, which produces one or more solutions. To find solutions, a solver combines **search** and **deduction**.

1. A systematic search through the space of partial assignments makes guesses about variable assignments. The assignments are done incrementally to a subset of the variables and the solver must backtrack if the current assignments cannot lead to a solution.
2. Constraint propagation is the way to make deductions based on constraints and current domains. This is usually recorded as reductions in domains so that when a variable has a single value left in its domain, that is part of the solution.

1.2 Constraint optimisation problem

$$\text{COP} = \text{CSP} + \text{an objective function} \tag{1}$$

The constraint optimisation problem adds an **objective function** to a normal CSP. The goal is then to maximise or minimise the objective function. The goal is then to find a solution where all constraints are satisfied *and* the objective is optimised. Examples of these problems include minimising time taken in scheduling problems, or maximising profit. This gives four components to the constraint problem.

1. **Given** parameters
2. **Find** decision variables
3. **Such that** constraints
4. **Min/Maximising** objective

1.3 Problem classes

A problem class describes a family of problems, related by a common set of **parameters**. Examples of a problem class is the n-queens problem class, or sudoku problem class. The parameters are the same as each problem instance in the class has the same rules.

It is important to note that constraint solvers solver problem **instances**. An instance is specified by instantiating the parameters of a problem class to particular values. In the sample of sudoku, a sudoku problem instance is the instantiation of the filled in hints given on the cells. Further, an individual CSP/COP represents a problem instance.

1.4 Constraint representation

1.4.1 Table constraint

The table constraint is the most basic constraint available. It consists of listing the satisfying combination of assignments for all constraints and is known as the **extensional** representation. This is the most basic constraint as any constraint can be modelled in a table.

For example given two variables

$$X \text{ with domain } 1, 2, 3$$
$$Y \text{ with domain } 1, 2$$

and the constraint $X > Y$, the table constraint would look as follows:

Table(X, Y)
$\langle 2, 1 \rangle$
$\langle 3, 1 \rangle$
$\langle 3, 2 \rangle$

The issue with this basic constraint is that is can become both cumbersome and not practical. In an example of sudoku, where an AllDifferent constraint has to be imposed on 9 variable, the table would build to be 9! tuples large. The space to store the table tuples is not the only issue, as having to search through all the tuples would also take a significant amount of time. Because of this issue of combinatorial explosion, **intensional** representations were designed.

1.4.2 Intensional constraints

Rather than explicitly list all satisfying variable assignments, constraint solvers typically use intensional constraints to represent constraints like AllDifferent. This is done by the constraint solver by

providing a library of commonly-occurring constraints that can be specified much more concisely.

However sometimes, the table (extensional) constraints must be used because it is the only sensible option, especially in examples where the constraint does not have an obvious algorithmic representation. This typically happens when the variables or domains are not numeric or easily represented as computing terms.

1.5 Constraint languages

Constraint programmers often do not want to work directly with CSP/COPs as they can be both large and too low-level to work with. Instead, the CSP/COPs are written as constraint programs (models) in constraint languages. The program/model is essentially a recipe to follow that reproduces a CSP/COP.

In a constraint language, decision variables and their domains must be declared. This is often done with arrays of variables, which allows for iteration to model problem classes. Further, constraint models written in constraint languages specify the problem *class* not the problem instance. The problem instances can be provided later as an instantiation of the parameters that the model specifies. Other common features in constraint languages include:

- Extensional constraints
- Equality, disequality, inequality
- Operators to build constraint expressions, for example +, -, AND, OR etc. These constraint expressions are then represented intensionally.

The model written in constraint languages can be parameterised to represent a problem class. This is done by giving values for the parameters to obtain an instance which corresponds directly to a CSP/COP. The CSP/COP can then be passed to a constraint solver to get the solution to that instance.

1.6 Crystal maze example

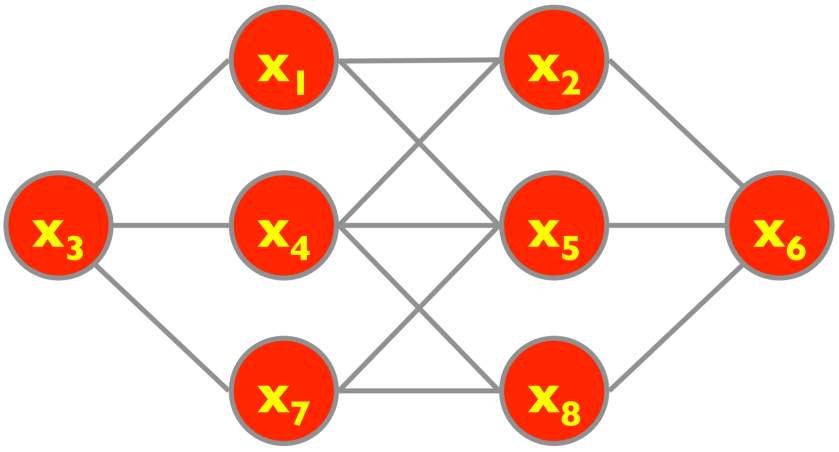


Figure 3: The crystal maze puzzle.

In the crystal maze puzzle, we must find $x_1 ... x_8$ with each domain $\{1, 2, 3, 4, 5, 6, 7, 8\}$ such that no adjacent nodes have adjacent values. For example, in the simple case of Constraint(x_1, x_2), the table constraint would look as follows:

Table(x_1, x_2)
$\langle 1, 1 \rangle$
$\langle 1, 3 \rangle$
$\langle 1, 4 \rangle$
...
$\langle 2, 2 \rangle$
$\langle 2, 4 \rangle$
...
$\langle 8, 8 \rangle$

Table 1: 50 possible combinations for each variable pairing that is connected.

The table initially is populated with 64 possible constraints for all values in the domain of x_1 and x_2. However, the constraint that says all connected variables cannot be assigned consecutive numbers removes some of the value assignments. The same constraint can be expressed in an intensional representation with the following equation:

$$|x_1 - x_2| \geq 1 \tag{2}$$

This works as it states that the values of x_1 and x_2 must have an absolute difference greater than 1. To be able to use the intensional representation, the constraint solver has to support the absolute value operator, subtraction operator and greater than constraint.

2 Modelling

2.1 Abstract constraint models

Constraint modelling is often considered the step to map a well-defined problem statement in a constraint language into a solver-independent constraint model for a constraint solver to solve and find solutions to. Solvers like Savile Row can take a solver-independent model and convert it into its own solver-specific model. However, when viewing constraint models *abstractly*, it can be seen that many combinatorial problems that we wish to tackle exhibit some *common features*.

Many problems require the programmer to find objects such as:

- (Multi-)sets
- Relations
- Functions

The abstract constraint model can be written in terms of there patterns. However, the patterns are typically not supported directly by constraint solvers (unlike intensional constraints. As such, the pattens need to be modelled as constrained collections of more primitive objects.

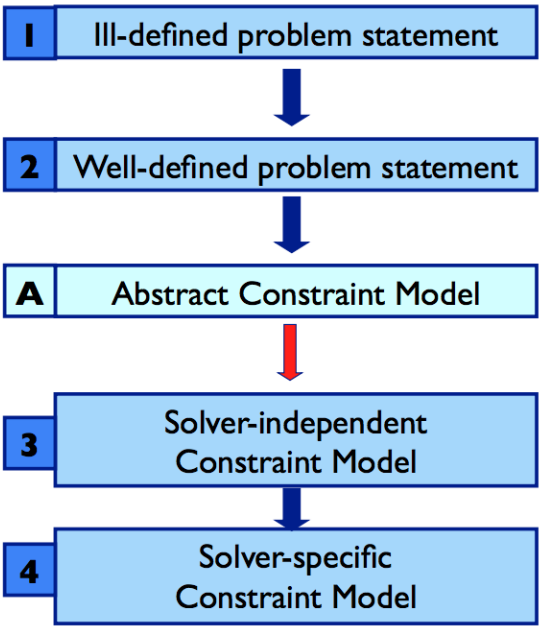


Figure 4: Steps to model a problem for a particular constraint solver.

This way, the corresponding **modelling patterns** for representing and constraining these combinatorial objects can be developed and effort is reduced when modelling new problems. Writing abstract constraint models gives a much richer set of types of decision variables, such as sequences, sets and functions and allows the programmer to extract the common features and patterns among different problems. Constraints for these variables and the objective function can also use operators on these new types of objects, such as set union, set relation etc. Finally, the patterns can be combined to model more compex problems.

2.2 Sequences

A sequence is defined as an **ordered list of elements**. It simply has elements in order and repetition is allowed. In the simple case, a sequence in of **fixed length**, or is **bounded** by a maximum length. The trickiest case for sequences is when the length is **unbounded**. Unbounded sequences are a problem because solvers need to use a finite-domain to model the CSP. In these cases, the programmer must be smart to find a bound by reading the problem carefully.

- 0, 1, 1, 2, 3, 5, 8, 14
- Turn right, drive 1 mile, turn right, drive 0.5 miles, turn left

are examples of sequences. This pattern typically occurs in planning problems, where a sequence of actions is the solution to transform the initial state of the problem into a goal state. Other examples include Langford's problem and the Bombastic problem.

2.2.1 Fixed-length sequences

Fixed-length sequences are problems of the form:

- Given **n**
- Find a sequence of objects of length **n**
- Such that ...

For example the magic sequence problem (CSPLib 19):

- Given **n**
- Find a sequence **S** of integers $s_0, ..., s_n$
- Such that there are s_i occurrences of i in **S** for each i in $0, ..., n$.

For the problem instance $n = 9$, a possible solution would be $6, 2, 1, 0, 0, 0, 1, 0, 0, 0$

To model fixed-length sequences, the most straightforward model is to use an array of decision variables indexied $1...n$ where the domains are the objects to be found.

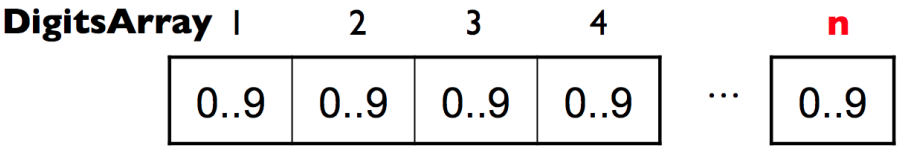


Figure 5: Array of decision variables 1 to n with their domain specified in an array.

As a constraint, this can be written as follows:

```
Forall i in 0..n .
    No. of occurences of i in S of i is S[i]
```

Depending on the constraint language being used, this statement may need to be expressed more or less concisely. For example, Essence Prime has a constraint `gcc(X, Vals, C)` which states that for each non-decision expression `Vals[i]`, the number of occurrences of `Vals[i]` in `X` equals `C[i]`.

Listing 1: The Magic Sequence problem modelled in Essence Prime.

```
language ESSENCE' 1.0

given n : int
find x: matrix indexed by [int(0..n)] of int(0..n)

such that
gcc(x, [i | i : int(0..n)], x)
```

- `x` is the array whose values we are counting and is represented by a matrix of decision variables
- `[i | i : int(0..n)]` is the list of values to count, which is a matrix of non-decision variables. The syntax here is a comprehension to produce the values `[0,1...,n]`
- `x` is used a second time here as the number of occurrences of each value, as it is the same as the value that is being counted

2.2.2 Bounded-length sequences

Bounded-length sequences are similar to fixed-length sequences, except that the length cannot exceed the maximum. The problems come in the form:

- Given **n**
- Find a sequence of objects of length **at most n**
- Such that ...

The Kiselman Semigroup Problem (KSP) is an example of a problem with infinite domain, but can be bound to a maximum length depending on the problem instance. The problem states that:

- Given n, a positive integer
- Find a sequence of integers drawn from $1..n$
- Such that between every pair of occurrences of an integer i, there exists an integer greater than i and an integer less than i

An example solution for $n = 3$ would be $2, 3, 1, 2$. In the definition of this problem, it can be seen that it is not finite. To *Find a sequence of integers drawn from* $1..n$, the sequence can be infinitely long. This is a problem because we want to map the abstract constraint model down to a finite-domain CSP and this is not possible with an infinite domain in the abstract model. To deal with this, we must derive a finite domain for KSP. Notice that there can be at most 1 occurences of 1 and n. Likewise:

- At most 2 occurences of 2 and $n - 1$
- At most 4 occurences of 3 and $n - 2$

From this pattern, we can derive a maximum sequence length of $\sum^{n} 2^{n/2} - 1$ pairs, which gives a finite domain for the sequence variable. Finally, because we have set a maximum length to the sequence, we may have to use **dummy variables** in solutions that are less than the maximum length, but still valid solutions.

We have to be careful when using dummy variables, especially if the value of the dummy exists in the domain of the variables as this will create **equivalence classes** of assignments. In that case it is impossible to tell if the value assigned to the variable is a real value or a dummy value. The solution to deal with this is to choose a **canonical element** from each class. For example, saying all 0s must appear at the end of the sequence if the dummy value is 0. Then the constraint will make sure to reject all other equivalences. Introducing equivalances during modelling is something that happens very often and one must be aware of this happening to use appropriate measures to counter it.

2.2.3 Unbounded sequences

For some problems, the issue of infinite domains cannot be dealt with in a smart way. This usually happens when a bound cannot be derived, or the derived bound is too weak to be useful. Problems with infinite domains are common when modelling planning problems, for example how many steps are needed to evacuate the building? Or how many actions it neeeded to reach the goal state?

Though inefficient, the solution to this is to solve a series of CSPs by incrementally increasing the length of the sequence. Moreover, this method allows the solver to find a solution with the shortest possible sequence.

2.2.4 Permutations

Some problems involve finding a sequence of elements where:

- The elements in the sequence are known
- The arrangement of elements is not known

The goal is then to find a **permutation** of the sequence that satisfies the constraints. The Travelling Salesman Problem (TSP) is an example of finding permutations, as the network between the cities and distances are all known.

2.2.5 Viewpoints

A **Viewpoint** is a choice of variables and domains sufficient to characterise the problem. In other words, if there is an assignment to the chosen variables, it can be read off a solution to the problem. Note this doesn't say anything about constraints.

For example, when looking for permutations, we can explore two different viewpoints:

1. Assume that the elements of the permutation are distinct. This leads to a fixed-length viewpoint.

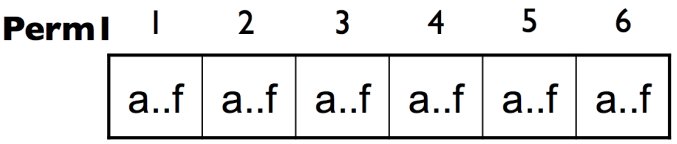


Figure 6: First viewpoint for a permutation which contains elements $a, ..., f$

2. Alternatively, we know the elements appear in the sequence, so we can index by those elements. The domain values now represent the position in the sequence an element is in.

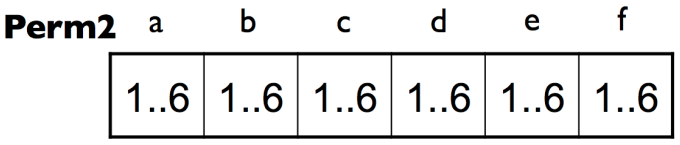


Figure 7: Second viewpoint for a permutation, using indices instead of elements.

Now there are two different viewpoints to choose from and this will depend on the kind of constraints needed to find a solution. For example, if the constraint is *a and b must be adjacent*, then the second viewpoint it much easier to model with the constraint `| Perm2[a] - Perm2[b] | = 1`. On the other hand, if the constraint is *The first three letters of the sequence must form an English word*, then it is easier to model the constraitn using the first viewpoint.

2.3 Sets

A set is a collection of *distinct* objects not arranged in any particular order. The items in a set must be unique (no duplicates). Examples of set would be $\{1, 2, 3\}$ or {red, green, blue}.

2.3.1 Fixed-cardinality sets

Consider the following simple problem class:

- Given n and s
- Find a set of n digits that sum to s

There are two ways to represent this problem, the **explicit representation** and **occurence representation**.

2.3.2 Explicit representation

For the explicit representation, an array E is introduces to hold the decision variables indexed by $1..n$, each with a domain of 0..9. The constraint for the problem is then simply:

```
AllDifferent(E)
Sum(E) = s
```

However, the issue of equivalence classes once again comes up here. The two sets {1,3,5,7} and {7,3,5,1} are equivalent as the order does not matter in a set, so a way to deal with this equivalence must be introduced.

1	**2**	**3**	**4**
1	3	5	7

Table 2: Explicit array representation of the set {1,3,5,7}

1	**2**	**3**	**4**
7	3	5	1

Table 3: Equivalent array reprensentation of the set {7,3,5,1}

A simple way to doing this would be to use ascending order as the canonial element from each class. Then the `AllDifferent(E)` constraint will be replaced by `E[1] < E[2] < E[3] < E[4]`.

2.3.3 Occurrence representation

Another way to represent sets is with an occurrence representation. This never introduces equivalence classes. It works like an array of binary switches. Following the above example, an occurrence representation O would be an array of 0/1 decision variables indexed by 0..9:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0,1	0,1	0,1	0,1	0,1	0,1	0,1	0,1	0,1	0,1

Now we simply need the two constraints:

```
sum(O) = n
O[1] + 2O[2] + 3O[3] + ... + 9O[9] = s
```

The first constraint only allows n 1s to be filled in the occurrence array and the second constraint is the constraint for the sum of digits.

Both explicit representation and occurrence representation have their advantages and disadvantages. For some problems, an explicit representation may introduce equivalence classes and be more complicated to specify a constraint. For example, if we wanted to say *if 5 is in the set, then so is 4*, this would need a constraint that takes more computation in the explicit representation.

Listing 2: Explicit representation

```
Forall j in 1..n .
    If (E[j] = 5)
        Exists i in 1..j-1 . E[i] = 4
```

Listing 3: Occurrence representation

```
(O[5] = 1) -> (O[4] = 1)
```

It should be noted that a combination of explicit and occurrence representations can be used to model problems, for example the intersection of two occurrence representations is an explicit

representation. In other words, the two are not mutually exclusive. Obviously, constraints on the original sets have to be modelled appropriately.

2.3.4 Set constraints

As sets appear frequently in constraint problems, it is useful to look at how common constraints such as intersection, union and subset are modelled. Any complex constraint can be decomposed to a simple constraint, each of which has at most one operator and possible one equality sign. For example, the complex constraint

$$(A \cup B) \subseteq (C \cap D) \tag{3}$$

can be decomposed into three constraints

$$X = A \cup B \tag{4}$$
$$Y = C \cap D \tag{5}$$
$$X \subseteq Y \tag{6}$$

2.3.5 Set intersection

The set intersection constraint can be modelled with both the explicit and occurrence representations. Take the simple constraint $A \cap B = C$ where A and B are sets of digits (domain 0..9), an occurrence representation would model it as the following three arrays

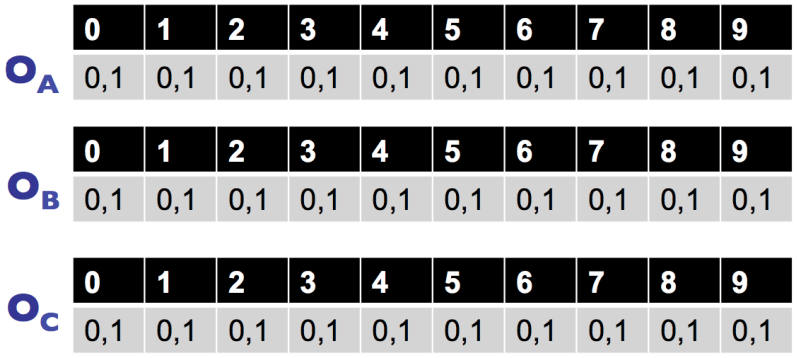


Figure 8: Three arrays of switches that map to A, B and C.

with the constraint

```
1  Forall i in 0..9 .
2    O_C[i] = O_A[i] × O_B[i]
```

To use explicit representation here would be more complicated and require more complex constraints.

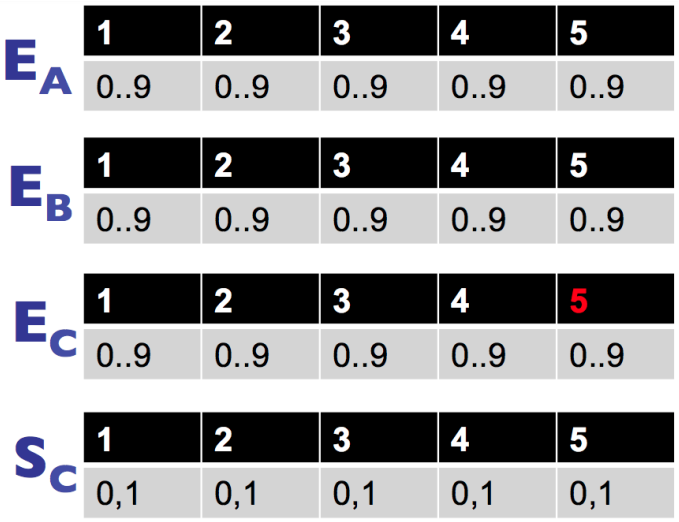


Figure 9: Arrays for explicit representation of set intersection. Cardinality of 5.

Each array represents the 5 values in the sets A, B, C, but the constraint to expresss the intersection is more complicated. Furthermore, it requires an extra marker array $\mathtt{S_C}$.

```
1  Forall d in 0..9 .
2    Forall i in 1..5 .
3      If (E_A[i] = E_B[i]) && (E_B[i] = d) .
4        d must appear in E_C[i]
```

2.3.6 Set union

The same arrays as in set intersection can be used to model the set union $A \cup B = C$ simply with a different constraint for an occurrence representation.

```
1  Forall i in 0..9 .
2    O_C[i] = (O_A[i] = 1 ∨ O_B[i] = 1)
```

Again this will be more complicated using an explicit representation which will be omitted here.

2.3.7 Set subset

2.4 Multisets

A multiset (sometimes referred to as "bags") is like a set except that duplicate values *are* allowed. They are also not arranged in any particular order and are typically characterised by the number of times each obejct occurs in the multiset. The multiset pattern occurs in places like the packing problem, for example a bag with a number of coloured balls.

Like sets, we must bound the domain of a multiset to be finite. For example,

- Let S be a finite set of size n.
- The domain "set of S" comprises of every set whose elements are members of S. This domain is the **power set** of S with size 2^n.

- The domain "multiset of S" will comprise of every finite multiset whose elements are all members of S. This will have an infinite domain as long as S is non-empty due to infinite repetition.

To ensure that the multiset of S is finite, we must bound either the total number of elements in the multiset, or bound the number of occurrences of each value.

2.4.1 Explicit representation

2.4.2 Occurrence representation

In the occurrence representation of multisets, instead of using 0 or 1 as values in the array, the array is extended to allow multiple occurrences.

2.5 Relations

Relations are an assignment of truth values to tuples of values. For example, in the set $P = \{\text{Bill}, \text{Bert}, \text{Tom}\}$, a binary relation *like* can be defined between two people from the set P. The cross product $P \times P$ can be used to get all tuples of the set P. We might assign the tuple <Bill, Bert> and <Bert, Tom> to be true and false for all other combinations. To visualise an occurrence representation of a relation R between two sets, a table can be created.

		B		
		2	3	4
A	1	0,1	0,1	0,1
	2	0,1	0,1	0,1
	3	0,1	0,1	0,1

Figure 10: Occurrence relation between two sets $A = \{1, 2, 3\}$ and $B = \{2, 3, 4\}$

2.5.1 Projection

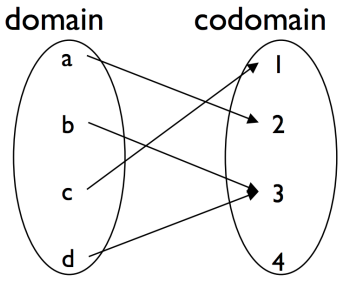
A common operation on relations in a projection. A relation can be projected onto one or more of its arguments, which results in a relation of reduced arity (less arguments/operands). For example in the set $P = \{\text{Bill}, \text{Bert}, \text{Tom}\}$ with the relation *likes* on $P \times P = \{<$ Bill, Bert $>, <$ Bert, Tom $>, <$ Bert, Bill $>\}$, the projection of *likes* onto Bert gives {Tom, Bill}.

2.5.2 Other representations

From simple binary relations, relations can be extended to k-ary relations. This is best thought of using a k-dimensional matrix. For exmaple in the ternary case of $A \times B \times C$, we can introduce a 3D matrix indexed by elements of A and B and the size of C.

2.6 Functions

A function f is a binary relation on two sets: a domain and a **codomain** (aka range). It has the property that each element of the domain is related to at most one element of the range which can be written as $f(x) = y$ to mean that the image of x under the function f is y.



In a **total** function, every element of the domain has an image in the codomain while in a **partial** function, some elements have no image in the codomain. For an explicit representation of partial functions, dummy elements have to be introduced, preferrably outside of the range of codomain values to avoid equivalence classes.

2.6.1 Injections

The images of two distinct elements of the domain under an **injective** function are distinct. In other words, no two domain values map to the same codomain value. An `AllDifferent` constraint will need to be added for these cases.

A **partial injection** is more complicated as not all domain values have to be mapped, but mapped values must be distinct.

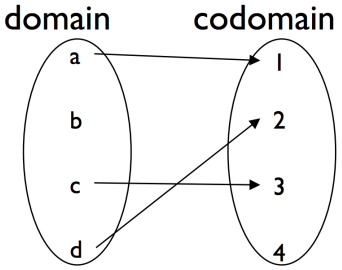


Figure 11: Domain and codomain mapping for partial injective functions.

Using an explicit model is messier here, but the occurrence representation is easy as we need the constraint that the sum of each row and col are both ≤ 1.

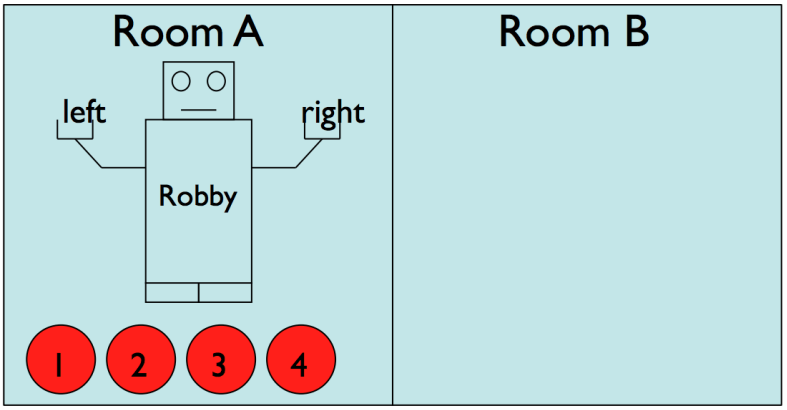
Furthermore, there are **surjective** functions, where every element of the codomain must be mapped to some element of the domain and **bijections**, which is both an injection and surjection.

2.7 Nested structures

Though we have explored many methods to model combinatorial objects, problems often require us to find one combinatorial objected nested inside another, for example a set of sets or a sequence of functions.

2.7.1 The Gripper Problem

In the gripper problem, the robot must move the 4 balls from room A to room B. The goal is to have all balls in room B and the robot has three actions: pick up, put down and move.



Since the robot has two hands, it can pick up or put down two balls in a single step. However, when he moves, he can't pick up or put down balls when he is moving. Hence there are at most 2 actions per step. Naturally, this is characterised as a **sequence of sets** with a maximum cardinality of 2.

2.7.2 Nesting inside sequences

Recall how fixed-length sequences of size n can be represented as an array of decision variables indexed from $1..n$ where the domains are the objects to be found.

0	**1**	**2**	**3**	**...**	**n**
0..9	0..9	0..9	0..9	...	0..9

In the case of nested sequences, we can no longer use a single variable at each index to represent the object at that position because one variable is not enough to represent our set, function or relation. A simple solution is to extend the dimension of the array. By having more dimensions, we can now have a column of variables for each index to represent the nested structure.

2.7.3 Nesting inside sets

Nested structures inside sets is being asked to find a set of some other object. Both the model of the outer type and model of the inner type (explicit/occurence) must be specified here, unlike in the case of nested sequences, where only one type was needed.

Consider the following problem class:

- Given m, n
- Find a cardinality-m set of sets of n digits such that ...
- There are three possibilities for models:
  1. An occurrence representation
  2. Outer: Explicit. Inner: Occurrence
  3. Outer: Explicit. Inner: Explicit

Nested sets: occurrence representation
It is not feasible to introduce an occurence array indexed by the possible set of n digits as this is not scalable. The size of the array would need to encompasss every possible set of size n digits which would be too large.

{1, 2, 3}	{1, 2, 4}	{1, 2, 5}	...
0,1	0,1	0,1	...

Because of this, nested outer layers are typically represented explicitly.

Nested sets: Outer explicit
To express the outer layer explicitly, the dimension of the explicit array can be extended according to the representation chosen from the inner set. Of course, we have to be careful to make sure the elements of the outer set are distinct.

EO	1	2	...	m
0	0,1	0,1		0,1
1	0,1	0,1		0,1
...				
9	0,1	0,1		0,1

(a) Occurrence representation of inner set

EE	1	2	...	m
1	0..9	0..9		0..9
2	0..9	0..9		0..9
...				
n	0..9	0..9		0..9

(b) Explicit representation of inner set

Listing (4) Constraints for occurrence representation of inner set

```
1 Forall i in 1..m .
2   sum(col i of EO) = n
3
4 Forall i in 1..m-1 .
5   col i of EO < col i+1 of EO
```

Listing (5) Constraints for occurrence representation of inner set

```
1   AllDiff on each column
2
3   Forall i in 1..m-1 .
4     col i of EE < col i+1 of EE
```

2.7.4 Relations as sets of tuples

Previously, we viewed relations as an assignment of truth values to tuples of values. This can be represented as a set of tuples.

2.8 Symmetry

The concept of symmetry is a **structure-preserving transformation** which can be characterised as a bijection on assignments (i.e. a one-to-one mapping). Complete assignments are partitioned into equivalence classes where all members in every class is either a solution or no member is a solution. In other words, symmetry preserves solutions. If there is a solution, symmetrical instances will also have solutions and vice versa.

In constraint programming, we do not want to deal with symmetries as it can lead to a lot of wasted effort for systematic search. There are two common special cases of symmetry:

1. Variable symmetry - where the bijection can be characterised in terms of variables alone
2. Value symmetry - where the bijection can be characterised in terms of values alone

2.8.1 Variable symmetry

Variable symmetry can be expressed in terms of a bijection on the variables. For example in the 4-queens problem, the board can be flipped horizontally, which leads to another valid and symmetrical solution.

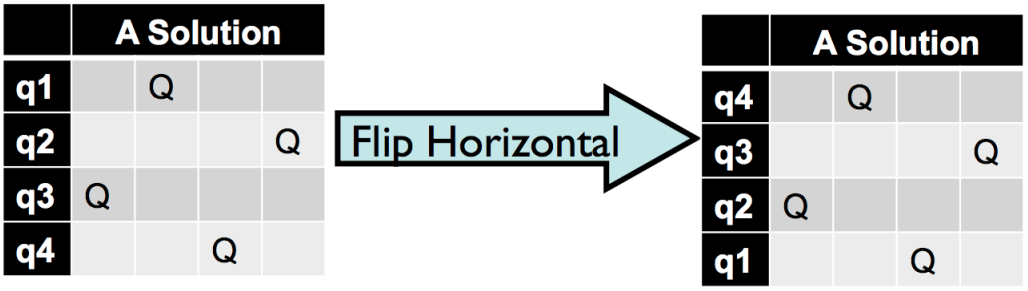


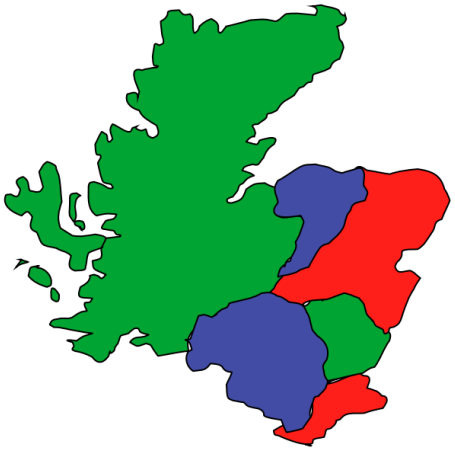
Figure 14: Two symmetrical solutions to the four queens problem.

- q1 → q4
- q2 → q3
- q3 → q2
- q4 → q1

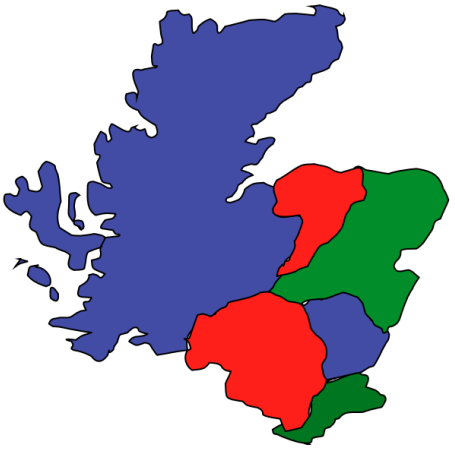
A bijection of values would capture the transformation of this particular solution - but not the general horizontal flip. Equivalently, a symmetrical non-valid solution is still non-valid.

2.8.2 Value symmetry

In value symmetry, the values assigned to variables can be permuted to give a symmetrical solution. An example of this is the graph colouring problem.



(a) Solution to graph colouring problem with map of Scotland.



(b) Symmetrical solution by permuting the value of the colours.

A symmetrical solution can be reached as shown in figure **??** by permuting the colours as follows:

- red → green
- green → blue
- blue → red

2.8.3 Consequences of symmetry

Assuming that an explored subtree contained no solutions, symmetries in the model can map this fruitless subtree into another. If the symmetry is not dealt with, a systematic search will explore

all symmetric variants of this subtree. It is important to remember symmetrical non-solutions are also non-solutions, and this is much more common than symmetrical solutions.

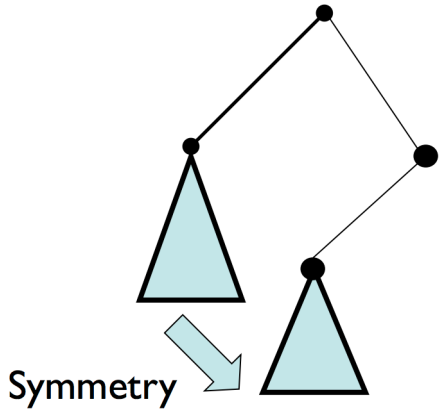


Figure 16: Symmetrical subtrees can map to another subtree later in the search.

2.8.4 Symmetry-breaking constraints

Knowing how symmetry is introduced during modelling can make some symmetry detection easy. There are two reasons that symmetry enters into a constraint model:

1. It is inherent in the problem by nature, for example in the n-queens problem as the solution is a square
2. It is introduced during the modelling process, especially for explicit representations.

Adding constraints can affect the symmetry, for example by disallowing some elements of an equivalence class of assignments. This is typically done by defining an ordering of elements. Ordering the elements distinguishes one element in each equivalence clas of assignments. If the symmetry-breaking constraint is effectively propagated, the search tree can be pruned drastically. Valid symmetry-breaking constraints leave at least one element in each equivalence class of assignments, so they throw away solutions to the original problem. This is fine as we can recover those solutions by applying the symmetry to the solutions allowed. Moreover, if enough symmetry-breaking constraints are added, *all* symmetry can be broken.

Typically, symmetry-breaking constraints are formulated by defining a canonical solution and adding constraints to choose it. In other words an ordering on variables and an order on solutions is needed.

2.8.5 Lex-leader method

To define an ordering, it is often best to use a **lexicographic** ordering, for example

$$\{a, b, c, d\} \leq_{\text{lex}} \{b, d, c, a\} \tag{7}$$

The lex-leader method is a way of applying lexicographic ordering to break all symmetry at the cost of a large number of constraints needed. It does so in two simple steps:

1. Choose an ordering on the variables
2. Add one lexicographic orderin constraint per symmetry

Although a huge number of constraints might be needed, sometimes the lex constraints collapse into simpler forms (like in the explicit set representation). A large number of constraints is needed because one constraint is needed for each possible permutation of the symmetrical transformation

of the problem instance. The compromise is to only use a subset of the lex-leader constraints. For example, given the array

A	B	C
D	E	F

with row and column symmetry, an ordering on variables ABCDEF can be chosen. With the lex-leader method, the following constraints have to be defined for the various types of symmetry that exist.

- ABCDEF $\leq_{\text{lex}}$ DEFABC - Swapping rows
- ABCDEF $\leq_{\text{lex}}$ ACBDFE - Swapping last two columns
- ABCDEF $\leq_{\text{lex}}$ DFEACB - Swapping rows and last 2 columns
- ...

This prevents it from being possible to obtain a "smaller" assignment that the one currently defined by applying symmetry. Depending on the problem, this could require a factorial number of constraints, which is too many and unscalable. This is where the Lex^2 ordering comes in.

2.8.6 Lex^2 ordering

A significant fraction of the lex-leader constraints for row/column symmetry can be represented by lex ordering only the rows and columns. However, it is not longer guaranteed to remove all symmetry in the problem. For example, AD ¿ BE BE ¿ CF ABC ¿ DEF are the Lex^2 constraints, which can be found in the full lex-leader constraints.

2.9 Viewpoints

The fundamental formulation to any constraint model is the selection of a viewpoint. A viewpoint is defined as a set of decision variables and domains sufficient to characterise the problem. The rest of the model and constraints follows from this choice of variables and domains.

Selecting a good viewpoint is essential to constraint modelling as a wrong choice can be writing the constraints very awkward, which likely leads to poor solver behaviour. Additionally, a good viewpoints makes expressing some of the constraints easier, but maybe not all constraints, so one viewpoint can be saved and auxiliary variables used to express the remaining constraints.

2.9.1 Auxiliary variables

Each variable in a CSP represents a choice that must be made to solve the problem being modelled. Because the original viewpoint is sufficient to represent all choices in the original problem, these **auxiliary variables** are extra in the sense that they are not needed to characterise the problem. In other words, the axuiliary variables are separate from the viewpoint that express constraints not in terms of the variables in the viewpoint.

As the choices that the auxiliary variables represent must also be represented by the original viewpoint, we must ensure that the two sets of variables are consistent by introducing **channelling constraints**.

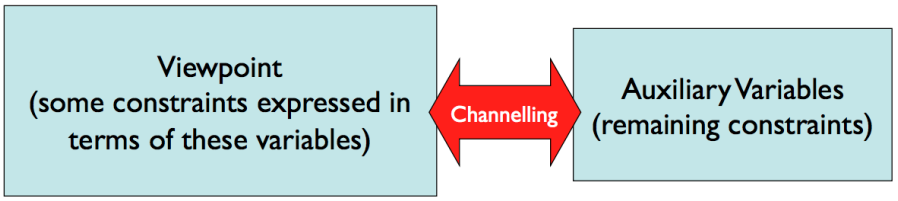


Figure 17: Auxiliary variables connected with channelling constraints to the original viewpoint.

Sometimes it is useful to formulate a model with 2 or more viewpoints to more easily express certain constraints. To add channelling constraints to link the auxiliary variables and original viewpoint, often a variable in one viewpoint has to be mapped or related in some way to the variables in the other viewpoint. Sometimes, extensional constraints work better here than intensional constraints.

2.10 Branching variables

Given a viewpoint and axuiliary variables, it is usually not necessary to assign all the variables to get a solution, as they may be redundant. The subset of the variable we choose to search over are the **branching** variables. It must be ensured that assigning the selected branching variables is sufficient to produce a solution. This is a choice that is made, especially when there are two viewpoints. The criteria for choosing which variables is to try and maximise constraint propagation.

2.11 Implied constraints

Implied constraints are logical consequences of the initial specification problem, when there is a chain of reasoning that leads to a deduction of implied constraints from the initial constraints. It is important to note that in theory, if constraint propagation was powerful enough, the solver would be able to see all these logical consequences, however, this is not the case in practice.

The constraints programmer must add implied constraints explicitly, to help reduce the amount of search needed in the solver. This needs to be done by hand because the generation of implied constraints involves complex reasoning steps which are difficult to automate. Further, programmers have to learn to do this by experience, as it is often the case implied constraints are missed because it logically does not need to be written explicitly.

There is a difference here between implied constraints and **redundant** constraints. The latter is also implied by the initial problem specification, but does not reduce search. These constraints should be avoided when possible as they add overhead to the solver and slow down the solving process.

Finally, symmetry-breaking constraints are *not* implied, as the CSP obtained by breaking-symmetry is not equivalent to the original problem in that it has a different set of solutions. In fact, when we break symmetry, there are new implied constraints that can be inferred. Often, the most powerful implied constraints are derived from symmetry-breaking, because more information can be inferred about the problem and therefore a stronger implied constraint can be written. For example, in the `AllDifferent` constraint on the explicit representation of sets

1	**2**	**3**	...	**n**
0..9	0..9	0..9	...	0..9

the symmetry-breaking constraint

$$E[1] \leq E[2] \leq E[3] \leq ... \leq E[n]$$

introduces a new implied constraint

$$E[1] < E[2] < E[3] < ... < E[n]$$

making the original `AllDifferent` constraint redundant.

Lex^2 ordered BIBD means both rows and columns are ordered as binary numbers. It can be seen that the initial row and column have 0s initially, so we can have stronger constraints defined. - Cannot swap row/cols where a 1 is before a 0 in the first row/col - Split into blue/red sections - Stronger constraints because the two exact sections must sum to a number, rather than the whole row must sum to a larger number

3 Search

3.1 Definitions

3.1.1 Systematic search

The idea of **search** is to make an educated guess among several available alternatives, but be prepared to undo the guess and try a different alternative if the guess does not lead to a solution.

More concretely, **systematic** search is the case where given sufficient time:

- If there is a solution, it will be found
- If there is no solution, the search space will be exhausted and the search will report that there is no solution

Typically, search is done through **partial assignments** of one or more decision variables which begins with an empty assignment and incrementally attempts to extend it into a solution. In this context, a **backtracking** search is one where if it is discovered that the current partial assignment cannot be extended to a solution (a dead end), then the algorithm backtracks over the last decision made and tries and alternative assignment.

3.1.2 Tree traversal

The search for a solution to a CSP can be easily viewed as exploring a tree where the root represents the CSP with no assigned variables and each choice corresponds to the branches of the tree.

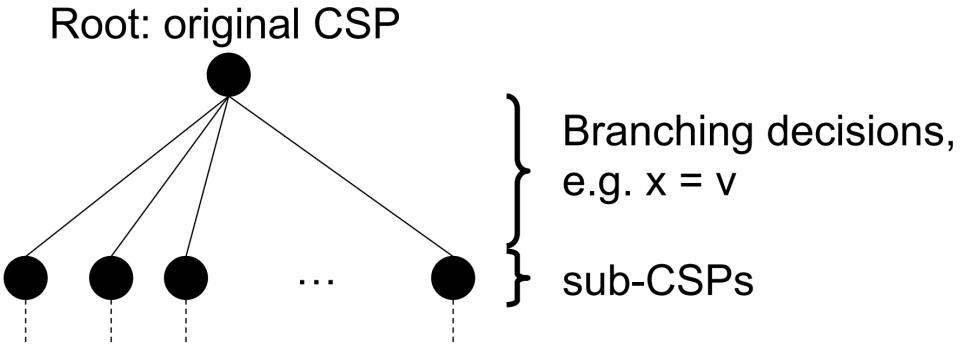


Figure 18: CSP search viewed as a tree.

The descending nodes correspond to sub-CSPs which are the original CSP with partial assignments. Furthermore there are two common branching styles used for tree traversal: **d-way branching**

and **2-way branching**.

D-way branching
In d-way branching, each branch under the parent node represents the assignment of one of the d domain values from the domain of a particular variable. For example, if $x \in \{1, 2, 3\}$, then there would be three branches which correspond to the assignment of x, one for each assignment.

2-way branching

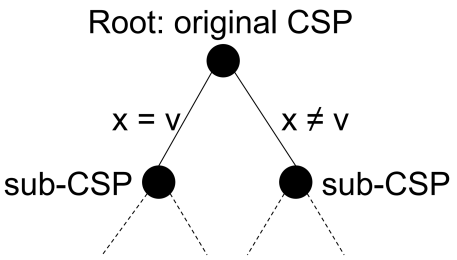


Figure 19: 2-way branching.

In comparison, 2-way branching divides the search space into two pieces, one where the assignment to a particular domain value is true, and the other when it is false. The left branch tries to extend the partial assignment with $x = v$ first and if no solution is found down that branch, v is removed from consideration entirely for the rest of the search.

3.1.3 Ordering

In a search problem, there are both **variable orderings** and **value orderings**. They specify the order in which the variables are assigned and the order in which the values for the variables are tested. These can be either fixed or dynamic, depending on the heuristic.

Variables that have yet to be assigned are called **future** variables and similarly the variables that have been assigned are called **past** variables.

3.2 Methods

3.2.1 Generate and test

Generate and test is a simple but very expensive method of solving a CSP. It is simply generating each **complete** assignment possible, then testing them all to see if any satisfy all the constraints.

This issue is further shown when trying to solve constraint *optimisation* problems because simply satisfying the assignments is not enough it does not guarantee the best assignment. Therefore in this case, generate and test must examine all complete assignments except in the case where we know the current solution cannot be improved.

Simply put, generate and test is a brute force method that only checks constraints *after* a complete assignment has been generated. It is systematic in that it is guaranteed to find a solution if one exists, however, due to its limitations in speed and memory it is never used in practice.

3.2.2 The backtrack algorithm

The backtrack algorithm improves on generate and test by incrementally extending partial solutions. Every time an assignment is make, the constraints are checked to make sure none are violated.

Algorithm 1 The backtrack algorithm

```
1: procedure BACKTRACK(depth)
2:     for d in D_depth do
3:         assign(x_depth, d)
4:         consistent = TRUE
5:         for past = 1 to depth - 1 while consistent do
6:             consistent = text(constraint(x_past, x_depth)
7:         if consistent then
8:             if depth = n then
9:                 showSolution()
10:            else
11:                BACKTRACK(depth + 1)
```

In the backtrack algorithm above, D_{depth} is used to mean the domain fo the variable being assigned at the level *depth*. After each assignment to a variable, all constraints with assigned variables are checked to ensure they are satisfied. If that all passes, then the algorithm either displays the solution or recurses. It is important to note that this algorithm will display *all* solutions.

In summary, the backtrack algorithm is a systematic search which guarantees to find a solution if one exists. It checks a constraint as soon as all of the variable that it constrains are instantiated. This allows it to spot dead-ends faster. In general, spotting dead-ends earlier leads to more search being saved.

3.2.3 Branch and bound

The branch and bound method is an augmented version of backtracking in order to find the best solution for optimisation problems. In this method, when a solution is reached, the value of the objective is recorded in α. Then in any further search, if the partial assignment cannot improve over the current best value α then we backtrack immediately.

In this case, there is a new function `calculateObjective()`, called the **bounding function** which calculates the lower bound on the objective from the current partial assignment. This assumes that any solutions extending the current partial assignment will have an objective calues equal to or greater than the current lower bound.

Algorithm 2 The branch and bound algorithm

```
1: procedure BranchAndBound(depth)
2:     for d in D_depth do
3:         assign(x_depth, d)
4:         objVal = calculateObjective()
5:         if objVal ¡ α then
6:             consistent = TRUE
7:             for past = 1 to depth - 1 while consistent do
8:                 consistent = text(constraint(x_past, x_depth)
9:             if consistent then
10:                if depth = n then
11:                    α = objVal
12:                    showSolution()
13:                else
14:                    Backtrack(depth + 1)
```

The branch and bound algorithm is also systematic, but further guarantees the *best* solution, not just any solution. It checks constraints as soon as all of the variables that it constrains are instantiated, just like in backtrack and maintains the value of the objective associated with the current best solution so that we can backtrack sooner done every search path.

3.3 Graphical representation

It is often useful to view a CSP as a graph rather than a series of constraints and variables as it allows us to guide heuristics and propagation.

3.3.1 Binary constraint graph

In a binary constraint graph, each node represents a variable and the constraints are represented by the edges between the nodes. This only works for binary constraints because a constraint between more than three variables would not be able to be represented by an edge between nodes.

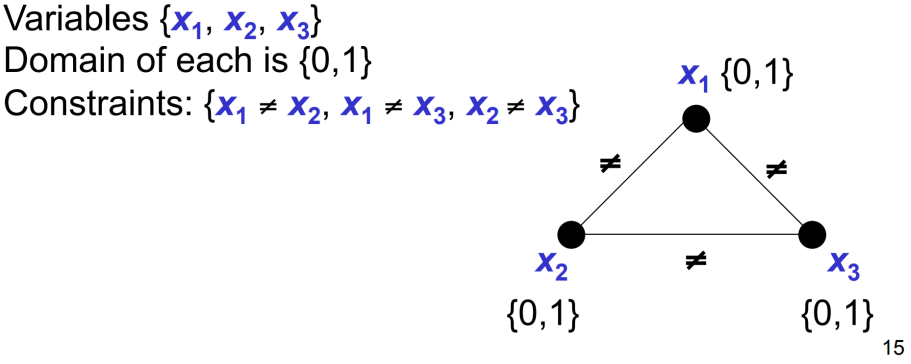


Figure 20: Example of a binary CSP and its graphical representation.

Because of the convenience of binary constraint graphs, we must find a way to turn non-binary instances into binary instances.

3.3.2 Dual representation

Dual representation is the method of transforming a non-binary CSP into a binary CSP. There are two simple steps to this conversion:

- Transform each constraint into a **dual variable** which has an associated domain of the allowed tuples
- Add **binary constraints** between any two dual variables that share an original variable. These constraints insist that the assignments to the two nodes that it connects agree for the shared original variables.

For example, given the non-binary CSP with the following variables and constraints:

$$x_1, x_2, x_3, x_4, x_5 \in \{0, 1\} \tag{8}$$
$$x_1 + x_2 \neq x_3 \tag{9}$$
$$x_2 + x_3 \neq x_4 \tag{10}$$
$$x_3 + x_4 \neq x_5 \tag{11}$$

We get the following tuples as the only valid assignments

$$y_1, y_2, y_3 \in \{\langle 0, 0, 1\rangle, \langle 0, 1, 0\rangle, \langle 1, 0, 0\rangle, \langle 1, 1, 0\rangle, \langle 1, 1, 1\rangle\} \tag{12}$$

However, tuples themselves cannot be used as domain elements as we think of domain values as **atomic**. Therefore, the tuples must be further transformed into an atomic value, the simplest method of which is to add up the values, which gives

$$y_1, y_2, y_3 \in \{1, 2, 4, 6, 7\} \tag{13}$$

The graphical representation is just that same as in a typical binary constraint graph. Each node will represent a dual variable y (an original constraint) and the edges between nodes corresponds to the new binary constraints which specify which of the original constraints share variables.

4 Propagation

Although backtracking in search is able to prune some parts of the search space, it is only looking backwards after an assignment has been made. However, we can be smarter than this to use constraints as a means of checking or pruning before actual assignments. This is known as **constraint propagation**, which deduces more information via a subset of the constraints. The deduced information can then be used to prune additional values from domains.

The idea is that local changes made form the basis for further deductions. Hence the result of a change is gradually *propagated* through the constraint network.

4.1 Consistency properties

A consistency properties holds when constraint propagation of a certain kind reaches a **fixpoint**, in other words when nothing new can be deduced. Consistency may be either

- **Local** - one (or a subset of) constraints/arc/variables
- **Global** - all constraints/arcs/variables

4.1.1 Local node consistency

Given a unary constraint $c(x_i)$, **local node consistency** holds if

$$\forall d \in D_i, c(x_1) \text{is satisfied when } x_i = d \tag{14}$$

For example, given the constraint $x_1 < 5$ when $D_1 = \{3, 4, 5, 6\}$, the node is not currently node consistent, so we can propagate $x_1 < 5$ to obtain $D_1 = \{3, 4\}$ which is node consistent.

4.1.2 Global node consistency

Global node consistency is simply an extension of local node consistency which holds if local node consistency holds for *all* unary constraints. It can be thought of as any single variable can be assigned a value from its domain consistently. Moreover, it can be enforced by simply enforcing node consistency for each unary constraint *once*.

4.1.3 Local arc consistency

As mention earlier, a constraint in a constraint graph is represented by an undirected edge between two nodes. For arc consistency, we consider splitting this up into **two directional arcs**

$$c(x_1, x_2) = arc(x_1, x_2) \wedge arc(x_2, x_1) \tag{15}$$

Local arc consistency is then defined as follows:

Given an arc $arc(x_i, x_j)$, local arc consistency holds if

- $c(x_i)$ and $c(x_j)$ are both node consistent
- For each $d_i \in D_i$, there is at least one corresponding $d_j \in D_j$ such that $c(x_i, x_j)$ is satisfied when $x_i = d_i$ and $x_j = d_j$.

The propagation to maintain local arc consistency is known as **arc revision**. For example, given the arc $arc(x_1, x_2)$ of the constraint $x_1 < x_2$ and the two domains $x_1 \in \{2, 11, 16\}, x_2 \in \{2, 5, 10, 11\}$, we can see that the values 11 and 16 in the domain of x_1 must be removed to maintain local arc consistency as they do not have any supporting values from the domain of x_2.

Supports is the concept of *evidence* that a domain value is consistent and therefore should not be removed. It is used in constraint propagation generally. In arc consistency, a support of a value in x_1 is a corresponding value in x_2 which satisfies the given constraint. For example, the values $\{5, 10, 11\}$ in the domain of x_2 support the value of $\{2\}$ in the domain of x_1 while $\{11, 16\}$ had no supports and were removed.

4.1.4 Global arc consistency

Global arc consistency holds if local arc consistency holds for *all* arcs. It can be thought of as any assignment to a single variable can be extended to an assignment to two variables consistently as global node consistency is a precondition.

However, unlike global node consistency, it *cannot* be enforced simply by enforcing local arc consistency for all arcs at once, as each time a domain has been pruned by one arc revision, connected arcs may no longer be consistent in the case that the only support is removed for some values.

4.2 Enforcing arc consistency

We know that it is not possible to enforce arc consistency in one pass in general, so algorithms are needed that an efficiently re-revise arcs as necessary. The goal is that after enforcing arc consistency, every arc in the problem is arc consistent.

4.2.1 AC1

AC1 is the simplest method of enforcing arc consistency which is equivalent to a brute force method. The idea is to iterate passes of the revision of every arc until there are no more changes that need to be made. This is obviously inefficient as all arcs are revised regardless of the possibility of them having become inconsistent.

4.2.2 AC3

Algorithm 3 The AC3 algorithm.

```
1: procedure AC3
2:     for i = 1 to n NC(x_1) do
3:         Q = allArcs(x_i, x_j)
4:         while not empty(Q) do
5:             Remove any arc(x_i, x_j) from Q
6:             if Revise(arc(x_i, x_j)) then
7:                 Add allArcs(x_h, x_i) to Q where h ≠ j
```

The algorithm of AC3 is explained following the pseudocode.

3. The queue is first initialised with all arcs in the instance.
4. The termination condition is when there are no arcs left to revise, because all arcs are consistent at this point.
5. The order of processing is unimportant in order to find a final result, but the ordering can impact how much work is required.
6. Assuming that REVISE() returns true if D_i has been changed and also assuming empty domains are caught.
7. If D_i changes, then we must re-revise arcs incident on x_i. Arcs that are already present to not have to be re-added as Q is a set. The only exception is to not add the arc $arc(x_j, x_i)$.

The reason $arc(x_j, x_i)$ does not have to be re-revised is because supports are **bi-directional**. That is to say if a value is supported on $arc(x_i, x_j)$, then it is supporting some value(s) on $arc(x_j, x_i)$. Conversely, if a value is not supported on $arc(x_i, x_j)$, then it is supporting no values on $arc(x_j, x_i)$ and therefore there is no need to revise $arc(x_j, x_i)$ after revising $arc(x_i, x_j)$.

Algorithm 4 Algorithm for the REVISE function

```
1: procedure REVISE(arc(x_i, x_j))
2:     changed = false
3:     for d_i in D_i do
4:         supported = false
5:         for d_j in D_j while not supported do
6:             if x_i = d_i, x_j = d_j satisfies c(x_i, x_j) then
7:                 supported = true
8:         if not supported then
9:             Remove d_i from D_i
10:            changed = true
11:    if empty(D_i) then
12:        fail()
13:    return changed
```

Note that the worst case for AC3 is when the binary constraint graph is fully connected, as every arc other than the one currently being revised as to be added to the queue again whenever the domain changes.

AC3 has a worst case time complexity of $O(ed^3)$ where e is the number of edges in the constraint graph and d is the maximum domain size. This is because each arc can only enter the queue at most d times and there are $2e$ arcs, so REVISE is executed O(ed) times.

4.2.3 AC4

The issue with AC3 is that it is sub-optimal as it adds all arcs back to the queue whenever a variable's domain has been changed. This often wastes considerable effort in adding arcs onto the queue whose revision will not lead to any domain pruning and is the root cause of its sub-optimal time complexity of $O(ed^3)$. There are two approaches to reduce this issue and produce an optimal arc consistency algorithm.

The idea is rather than having a corse-grained queue Q of arcs to revise, we have a fine-grained list of deleted values L. This way, when revising an arc, it knows which value was deleted. AC4 is an example of such a fine-grained algorithm.

In AC4, two additional structures are needed

- A **counter** is associated with *each* arc-value pair $\langle arc(x_i, x_j), d_i \rangle$ where d_i in D_i which records the number of elements of D_j that support d_i.
- A **set** of pairs $\langle x_i, d_i \rangle$ associated with each d_j in D_j which records the assignments for which d_j in D_j provides support.

This way, for each domain element, we know how many supports it has, and for which other elements it provides support. This leads to the key aspect of AC4, which is to maintain a table of deleted domain values M where $M[x_i, d_i]$ is 1 if deleted and 0 if otherwise and to maintain a list L of deletions to propagate. The list L contains individual value deletions rather than arcs.

The algorithm has two phases, initialise and propagate.

Initialise

- Process each $arc(x_i, x_j)$ once
- Enumerate support for each element in every domain D_i

- If not support for $x_i = d_i$, then it is pruned and the list of deletions L is initialised

Propagate

- Choose a deleted value to propagate from L
- Iterate over the assignments supported by the deleted values (S)
- Decrement their counters
- If the counter reachers 0 and the element is not already deleted, delete and add to L
- Repeat until arc consistency is established on all constraints (i.e., when L is empty)

AC4 has a worst-case time complexity of O(ed^2) where e is the number of edges in the constraint graph and d is the maximum domain size. O(ed^2) time is needed for both the initialisation and propagation phase. In initialisation, for each arc is a double-nested loop of length d. In propagation, there are at most O(ed) counters, each decremented at most d times. This is worst-case optimal, it always reached the worst case because of the initialisation of the data structures.

Although AC4 has a better time complexity compared to AC3, it has a greater space complexity, requiring O(ed^2) worst-case space complexity whereas AC3 only has a worst-case space complexity of O($e + nd$).

4.2.4 AC2001/3.1

The AC2001/3.1 algorithm adds the *Last* data structure to the AC3 algorithm. This structure contains the last support in D_j for $x_i = d_i$. This way, when revising the arc $arc(x_i, x_j)$ and checking the viability of d_i, the Last($arc(x_i, x_j), d_i$) can be checked to see if the last support is still present. If not, we only have to check for support in its successors.

There is a subtle difference here in AC2001/3.1 compared to AC4.

4.3 Forward checking

Forward checking is a combination of propagation and search. It is the fundamental way in which systematic constraint solvers work and follows approximately the following steps:

1. Guess an assignment
2. Propagate consequences of that assignment
3. If all is well, back to step 1

The basic idea is to revise arcs from every future variable to x_i once after an assignment to x_i, which enforces local arc consistency on a subset of the arcs. This helps spot dead-ends earlier to reduce the amount of search.

```
1: procedure ForwardChecking(depth)
2:     for d in D_depth do
3:         assign(x_depth, d)
4:         consistent = true
5:         for future = depth + 1 to n while consistent do
6:             consistent = Revise(arc(x_future, x_depth))
7:         if consistent then
8:             if depth = n then
9:                 showSolution()
10:            else
11:                ForwardChecking(depth + 1)
12:        undoPruning()
```

Notice that forward checking does not require checking against the past variables, because the arc to the current variable was already checked when the past variable was assigned. This makes it so when considering an assignment to a current variable, all possible assignments are guaranteed to be compatible with the assignment to the past variable.

Forward checking is complete and guarantted to find a solution if one exists. It is also guaranteed to explore a search tree that is *smaller than or equal-sized* to that of backtrack. This saves search compared to backtrack whenever there is a domain wipeout of a future variable. Finally, forward checking can be combined with branch and bound for optimisation problems to result in better information about the objective and better propagation of the current best value.

4.4 Heuristics

Heuristics for CSP/COPs are rules of thumb that can be followed, which usually is expected to lead to an improvement. There are two different operations which we can order heuristically when trying to solve a constraint problem:

- Variable assignment order
- Value assignment order

In particular, the perfect heuristic for value assignment will lead *directly* to a solution.

There are also two different forms heuristics can come in, **static** and **dynamic**. Static heuristics are ones in which the order is chosen before the search begins and is fixed for the entire search. Dynamic heuristics work the other way around where the state of the problem is examined and the best variable/value to assign next is decided based on the current state.

The variable ordering during search can have a significant effect on the speed of the search. In particular, inefficient variable ordering leads to lots of thrashing, where repeated failures occur for the same reason because of the variable order. The rationale is therefore to make the **most difficult** choice first, which is likely to have the most far-reaching consequences for the rest of the problem. This has the added effect of narrowing down easy choices.

4.4.1 Static variable order heuristics

There are three static variables heuristics we will look at, all three require a graph to represent the constraints called the **primal graph**. The primal graph has an edge between two variables iff they are in a constraint together. Unlike the constraint graph, where each edge represents one constraint.

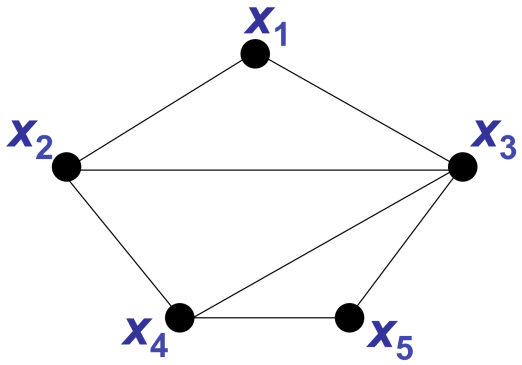


Figure 21: Example of a primal graph for the CSP with the three constraints $c(x_1, x_2, x_3), c(x_2, x_3, x_4), c(x_3, x_4, x_5)$

Maximum degree
The maximum degree heuristic orders the variables in decreasing order of their degrees in the graph. Any ties with the same order are chosen arbitrarily. This follows the rationale of choosing the most difficult choice first as variables that are part of a lot of constraints are more highly constrained and highly constrained variables may be the hardest choices.

Maximum cardinality
In maximum cardinality, the first variable is selected arbitrarily and the next variable is the variable connected to the largest group among those already selected. The idea behind this is after assigning some variables, we reason that the next hard choice is the one most constrained by the existing assignment as we need to make choices compatible with the existing assignment.

Minimum width
The minimum width heuristic uses an ordered graph, which is an arrangement of graph nodes into a fixed linear order. The **width** at an ordered node is defined as the number of arcs linking back to a previous node in the order. The width of an ordering is the maximum width at any node and the width of a graph is the minimum width of all its orderings. To use this heuristic, the minimum width of the graph and an ordering corresponding to this width.

4.4.2 Dynamic variable order heuristics

Smallest domain first
Smallest domain first selects the variable with the smallest number of values compatible with past assignments. This is dynamic rather than static because the selection is made based on the state *during* the search. As such, different branches of the search tree can have different variable orders.

Brelaz
The Brelaz heuristic selects the variable with the smallest number of values compatible with past assignments just like smallest domain first. The difference is that to break ties, the variable with the maximum degree in the constraint sub-graph of the future variables is chosen.

dom/deg
domdeg

4.4.3 Value assignment order

Min-conflicts asd

Geelen's value-ordering heuristics

4.4.4 Variable order

4.4.5 Value order

4.5 Maintaining arc consistency

While forward checking performs just sufficient propagation that it no longer needs to check backwards, we could do better by imbedding arc consistency algorithms into a search to enforce arc consistency. This leads us to the idea of **MAC** (Maintaining Arc Consistency).

The basic idea is to always make sure the partial assignments made continue to have global arc consistency. After every assignment, global arc consistency must be re-established. This gives the benefits of spotting dead-ends earlier than forward checking and is relatively efficient as long as an efficient arc consistency algorithm is used.

To do this, it is not necessary to revise all arcs at every node on a branch of the search tree. The trick is to think of assigning a value to a variable as saying that variable only has the domain with that one value and running a standard arc consistency algorithm on it. MAC also uses two way branching as it allows arc consistency to be established on the right branch after the value removal. After every left branch $x_i = d_i$, the queue is initialised with the consequences of removing all but d_i from D_i, which propagates these changes to re-establish arc consistency. The right branch is handled similarly.