How to build a constraint propagator in a weekend

Ian Horswill and Leif Foged
Northwestern University
ian@northwestern.edu, leif@u.northwestern.edu

Abstract

Constraint programming is an appealing, declarative programming technique that offers many attractive features for game AI. Unfortunately, the literature on building constraint satisfaction systems is somewhat daunting. In this paper, we will discuss the issues involved in building the simplest class of constraint solvers and discuss what kinds of problems they're appropriate for.

Introduction

Constraint programming provides a simple **declarative** mechanism for solving certain kinds of **configuration** problems. For example, if you're populating a level for a roguelike, you might not care which room has the boss, provided it's a room relatively near the exit; you might not care exactly how many enemies there are, so long as their numbers are in some reasonable range; and you might not care how much ammo you place, so long as it's appropriate for the number of enemies. These kinds of problems can be easily encoded as **constraint satisfaction problems**, and solved automatically by standard methods.

The use of constraint solvers for game AI was pioneered by Gillian Smith's work on constraint-based level design tools, and Adam Smith's work on automated design of game rule systems. These authors used off-the-shelf constraint solvers for their work. However, these solvers would be difficult to integrate into an existing game engine, particularly for a console. That said, constraint solvers – at least simple ones – really aren't that difficult to understand an implement.

In this tutorial, we'll talk about how to build a simple **finite-domain constraint solver** that can quickly solve configuration problems such as item-and-enemy placement, and talk performance optimization and issues involved in implementing it in different kinds of run-time environments.

In writing the code here, our priority has been to make the code **concise and pedagogical** rather than fast. So there are a number of cases where we will omit obvious optimizations so as to keep the code concise and easy to understand. In these cases, we will note the optimizations and discuss any trade-offs involved in implementing them. Please note also that the code given here was written off the top of Horswill's head without attempting to compile it, so it should not be taken as gospel.

Constraint satisfaction problems

A **constraint satisfaction problem** (CSP) is simply a set of **variables** you'd like to find values for, along with a set of **constraints** on what combinations of values are allowable.

Each variable has a specified set of candidate values, referred to as its **domain**. For the purposes of this article, you can think of a domain as a data type. We will focus on **finite domains**, which are essentially enumerated types (C `enums`). However, we will also talk a bit about handling numeric domains.

Constraints can theoretically be any **relationship that needs to hold** between the values of some set of variables, provided you can write code to at least test whether the relationship holds. Common examples are **equations**, **inequalities**, and limitations on **how many times** a given value can be used.

So when you specify a CSP, you specify:

1. A set of variables to find values for, along with their domains, and
2. A set of constraints on what combinations of assignments are valid.

As a very simple example, let's consider the following CSP: you want to assign a set of meetings to days of the week (Monday through Friday) subject to the constraints that:

- Certain meetings can't be the same day because they have the same people attending
- One particular meeting has to come after all the other meetings

We can represent this using a CSP with:

- One variable $v_i \in \{\text{Monday}, \dots, \text{Friday}\}$ per meeting.
- A set of constraints on the variables
    - Some pairs of meetings (e.g. meetings 3 and 5) have to be on different days: $v_3 \neq v_5$
    - The magic meeting (call it meeting 1) has to come last: $v_1 > v_2, v_3, \dots, v_n$

Some naïve algorithms and why they're slow

CSPs are fundamentally **search problems**, so the most basic algorithm is to **try all possible combinations** of values for variables until you find one that works:

```
for each possible assignment of values to the variables $v_1, v_2, \dots, v_n$
   if values of $v_1, v_2, \dots, v_n$ are consistent with the constraints
```

```
        return the values of $v_1, v_2, \dots, v_n$
    return failure
```

In practice, this gets done by choosing values for the variables one at a time. So it's functionally just a set of nested loops:

```
    for each possible value of $v_1$
      for each possible value of $v_2$
        …
          for each possible value of $v_n$
            if values of $v_1, v_2, \dots, v_n$ are consistent with the constraints
              return the values of $v_1, v_2, \dots, v_n$
    return failure
```

The obvious problem with this is that we may have to look at a very large number of variable assignments until we find one that works. If your problem has v variables over a domain with d elements, but only s solutions, then on average, you'd expect to have to run for d^v/s iterations to find a solution; that's unworkable except for very small values of v or extremely large values of s.

We can make this better by trying to do **early detection of obviously bad choices**. As an example of a problematic bad choice, consider running this algorithm on the constraint problem above. The first variable it will assign a value to is v_1, the magic meeting that has to come last. The algorithm will start by setting it to Monday, then Tuesday, Wednesday, etc. Unfortunately, Monday is an impossible value for this variable because all the other meetings would have to come before Monday, and we've assumed Monday was the earliest day. The problem is the algorithm will crank through every possible day for each of the other meetings, hoping to find some schedule that works, before it finally moves on and tries another day for the magic meeting. So in this case the algorithm will search virtually the entire search space before finding a solution – the worst behavior possible.

We can improve the algorithm by sanity checking variables as we assign them values. When we assign a variable a value, we first check whether it's obviously incompatible with the values of any other variables we've assigned:

```
    for each possible value of $v_1$
      for each possible value of $v_2$ consistent with $v_1$
        for each possible value of $v_3$ consistent with $v_1, v_2$
          for each possible value of $v_4$ consistent with $v_1, v_2, v_3$
            …
              for each possible value of $v_n$ consistent with $v_1, v_2, \dots, v_{n-1}$
                if values of $v_1, v_2, \dots, v_n$ are consistent with the constraints
                  return the values of $v_1, v_2, \dots, v_n$
    return failure
```

Now the algorithm will still try to put v_1, the final meeting, on Monday. But this time, when it tries to assign v_2 a value, it will find there are no values of v_2 consistent with $v_1 =$ Monday, so it will give up immediately and try the next value of v_1.

On the other hand, if our constraint wasn't that v_1 had to come after all other meetings, but just that it had to come after v_5, then it will still consider every possible schedule of meetings v_2, v_3, and v_4 before giving up and finding a new day for v_1. Not good.

The problem is that this algorithm only sanity checks a choice against **past choices**, not against the choices it will have to make in the **future**. What we want is some way of realizing when we choose a value for v_1 that it kills off all possible choices for some other variable, and undo that choice immediately. That's what constraint propagation does; it lets us compare current choices against possible future choices.

Constraint propagation

The basic idea of constraint propagation is to track a **set** of values for each variable, not just a single value. For the moment, we'll assume all the variables have finite domains, so the sets are just **bit masks** and the set operations are nice and simple. Here's the basic idea of constraint propagation:

- At the start of the algorithm, the variables can have any possible value.
- Choosing a value for a variable corresponds to narrowing its set to a single value.
- Each time we narrow a variable (rule out possible values for it), we **propagate** the restriction to other variables, ruling out any values for those variables that are no longer possible because of the restrictions introduced on the first variable. We'll talk about how this is done shortly.
- Any time a choice for one variable narrows another variable to the empty set, we know that choice was bad and try another one.
- We keep selecting values for variables until all variables are narrowed to a single value.

So the high-level outline of the algorithm looks like this. For each variable v_i, we store a set, v_i.values, which holds the set of values for the variable that haven't yet been ruled out. Unfortunately, the code for this gets unwieldy as a nested iteration:

```
for each $i$, $v_i$.values = all values
for each $v \in v_1$.values {       // Pick a value for $v_1$
    narrow $v_1$.values to just { $v$ }, narrowing other variables $v_j$ as necessary
    if no variable got narrowed to the empty set {
        for each $v \in v_2$.values {    // Pick a value for $v_2$
            narrow $v_2$.values to just { $v$ }; narrow other variables as necessary
            if no variable got narrowed to the empty set {
                // Try choosing a value for $v_3$
                … etc …
            }
            undo all updates to all variables we did in this iteration
        }
      }
      undo all updates to all variables we did in this iteration
  }
  return failure
```

So instead, we'll write it equivalently as a recursion. We have a basic procedure, Solve(), that initializes and then calls SolveOne(), which picks a single variable and value for it, then recurses to pick the next variable. When all variables have been narrowed, we're done. There's a lot of hand waving in this code, but it's a start:

```
Solve() {
   for each $i$, $v_i$.values = all possible values
   SolveOne()
}

SolveOne() {
   if all variables have exactly one value
      print the values and exit
   else {
     pick a variable $v_i$ with more than one possible value
     for each $v \in v_i$.values {       // Pick a value for $v_i$
        Narrow($v_i$, { $v$ })
        if no variable got narrowed to the empty set
            SolveOne()
        undo all the updates done in this iteration
     }
}
```

This is basically a correct algorithm. It just leaves some important implementation aspects unspecified:

- How do we determine whether one variable needs to be narrowed as a result of another variable being narrowed? That is, how do we actually do the propagation?
- How do we track the changes made to the variables so that we can undo them later?
- How do we actually backtrack to the last choice point when we discover that something's gone wrong?

We'll talk about each one of these in turn.

Implementing narrowing and propagation

This is the core of the algorithm. Each time we narrow a variable, we first check if we've reduced its possible values to the empty set. If so, then the system of constraints isn't solvable given the choices we've made so far, and so we "fail", meaning we jump back to the last choice we made and try a different choice, undoing all the variable updates we've done in the meantime. We'll talk about failing and unwinding the undo stack in the next two sections.

Assuming the variable hasn't been narrowed to the empty set, we tell the constraints applied to that variable in that its value set has shrunk; it can then update the other variables in the constraint appropriately:

```
Narrow(v, set) {
   newset = v.values & set    // assuming here that sets are just bitmasks
   if (newset == empty set)
      fail                    // we'll talk about how to implement this shortly
   if (newset != v.values) {
      v.values = newset
      for each c in v.constraints
         Propagate(c, v)
   }
}
```

The constraint then (somehow) computes how that restriction will affect any other variables in the constraint and narrows them appropriately:

```
Propagate(c, modifiedVariable) {
   for each v in c.variables
      if (v != modifiedVariable)
         Narrow(v, RemainingPossibilities(c, v))
}
```

This pseudocode is deeply hand-wavy. In practice, you implement constraints as a class hierarchy, where Propagate is a virtual function and each type of constraint has its own Propagate method. So we'll pick up with the details of Propagate methods when we talk about specific types of constraints.

Implementing backtracking

When you discover you've narrowed a variable to the empty set, the narrow operation has to fail and you need to return to the last choice point and try a different choice. Unfortunately, you can't do something as simple as returning an error code because Narrow() probably wasn't called by the procedure making the choice; it was probably called Propagate, which was probably called by Narrow, which was probably called by Propagate, etc. So we need a non-local exit mechanism. Unfortunately, there aren't any great alternatives for this in contemporary languages. Let's go through the available options.

Exceptions

The obvious choice is to treat failure as an exception. When you make a choice, you wrap the code in a try block:

```
SolveOne() {
  if all variables have exactly one value
    print the values and exit
  else {
    pick a variable $v_i$ with more than one possible value
    for each $v \in v_i$.values {        // Pick a value for $v_i$
      try {
        Narrow($v_i$, { $v$ })
        if no variable got narrowed to the empty set
          SolveOne()
      }
      catch Fail { }
      undo all the updates done in this iteration

}
```

And when you want to fail you just throw:

```
Narrow(v, set) {
  newset = v.values & set    // assuming here that sets are just bitmasks
  if (newset == empty set)
     throw new Fail()
  if (newset != v.values) {
    v.values = newset
    for each c in v.constraints
       Propagate(c, v)
  }
}
```

The problem with this is that exception handling is usually implemented by language designers with the assumption that it will only be used in exceptional circumstances; so it's usually slow. For example, under Windows, a process under debugging always sends a message to its debugger when it throws. That's potentially useful, although it makes debugging a backtracking algorithm glacially slow. But even when the process isn't running under a debugger, the implementation of throw still involves a system call; not good.

Unfortunately, at the same time language designers added exception handling to their languages, they typically also removed other mechanisms for non-local exit, so there aren't any really good mechanisms available.

Setjmp/longjmp

If you're programming in C or C++, you can use the low-level jump buffer mechanism, which was the old-school C way of doing non-local exit. The setjmp() call stores enough of the register set (SP, FP) to reproduce the state of the processor when it's inside of the setjmp() call. A call to longjmp() then restores that state, causing the processor to immediately "return" from setjmp for a second time.[1] The return value of setjmp() tells you whether you're returning from the original call to setjmp() or whether you're returning as a result of a call to longjmp(). The code for the constraint propagator then looks something like this:

```
Stack<jmp_buf> failStack;
jmp_buf fail;

SolveOne() {
   if all variables have exactly one value
      print the values and exit
   else {
     failStack.Push(fail)
     pick a variable $v_i$ with more than one possible value
     for each $v \in v_i$.values {      // Pick a value for $v_i$
        if (!setjmp(fail)) {
            Narrow($v_i$, { $v$ })
            if no variable got narrowed to the empty set
               SolveOne()
        }
        undo all the updates done in this iteration
     }
     failStack.Pop(fail)
    }
}
```

While Narrow looks like:

```
Narrow(v, set) {
   newset = v.values & set    // assuming here that sets are just bitmasks
   if (newset == empty set)
       longjmp(fail)
   if (newset != v.values) {
      v.values = newset
      for each c in v.constraints
          Propagate(c, v)
   }
}
```

[1] Functional programming enthusiasts will recognize this as a restricted version of call-with-current-continuation, supporting only upward continuations.

However, keep in mind that longjmp() will not interoperate with catch/throw, so you can't use it if you need exception handling for other things.

Return codes

The only other alternative is to go really old school and have both Narrow() and Propagate return a failure code and then scrupulously check the failure code every time to call it. That's both slower and more error prone than longjmp(), but it's pretty much what you're stuck with if you're writing in C# or Java. Then your code looks something like:

```
SolveOne() {
   if all variables have exactly one value
      print the values and exit
   else {
     pick a variable $v_i$ with more than one possible value
     for each $v \in v_i$.values {      // Pick a value for $v_i$
        if (!Narrow($v_i$, { $v$ }))
           return
        if no variable got narrowed to the empty set
            if (!SolveOne())
               return
        undo all the updates done in this iteration
     }
}

Narrow(v, set) {
   newset = v.values & set    // assuming here that sets are just bitmasks
   if (newset == empty set)
      return false
   if (newset != v.values) {
      v.values = newset
      for each c in v.constraints
         if (!Propagate(c, v))
            return false
   }
   return true
}

Propagate(c, modifiedVariable) {
  for each v in c.variables
     if (v != modifiedVariable)
        if (!Narrow(v, RemainingPossibilities(c, v)))
           return false
  return true
}
```

Implementing the undo stack

The last piece of functionality we need is the ability to undo updates to variables when we backtrack. This is straightforward:

- We keep a stack of saved variable/value pairs.
- Each choice point remembers how deep the stack before the choice. We'll call this the start of the choice point's stack frame.
- When we modify a variable, we push it on the stack with its old value.
- When we backtrack, we undo all the operations saved in the current choice point's frame, by popping saved values off the stack and undoing them, until the stack is at the depth it was before the choice.

We can optimize this by remembering what frame a variable was last saved in, and only saving the variable if it hasn't already been in this frame. The resulting code looks like this:

```
SolveOne() {
   if all variables have exactly one value
      print the values and exit
   else {
     int frame = undoStack.count;
     undoStack.currentFrame = frame;
     pick a variable $v_i$ with more than one possible value
     for each $v \in v_i$.values {       // Pick a value for $v_i$
        if (!Narrow($v_i$, { $v$ }))
           return
        if no variable got narrowed to the empty set
            if (!SolveOne())
               return
        while (undoStack.count != frame) {
            pair = undoStack.Pop()
            pair.var.values = pair.saveValues
        }
        undoStack.currentFrame = frame
     }
}

Narrow(v, set) {
   newset = v.values & set    // assuming here that sets are just bitmasks
   if (newset == empty set)
      return false
   if (newset != v.values) {
      if (v.lastFrame == undoStack.currentFrame) {
          undoStack.Push(v, v.values)
```

```
            v.lastFrame = undoStack.currentFrame
         }
         v.values = newset
         for each c in v.constraints
            if (!Propagate(c, v))
               return false
      }
      return true
   }
```

A basic set of classes

To simplify presentation, we didn't worry about modularity in the pseudocode above. Now let's look at what the code would look like in something closer to a real codebase. We'll outline a set of basic classes for implementing the code above in C#.

We'll start with the Variable class. This needs to keep track of its set of possible values, and implement methods for Narrowing and testing if a variable has been narrowed to a single solution. We will also put the undo stack into this class as a set of static members.

```
public class Variable {
    public UInt64 Values { get; private set; }

    // True if the variable has exactly one possible value
    // This is true when the bitmask is a power of 2
    public override bool IsUnique
    {  get  {  return (Values != 0) && ((Values & (Values - 1)) == 0); } }

    // Constraints involving this variable
    List<Constraint> Constraints { get; set; }

    public void Narrow(UInt64 set) {
       var newValues = set&Values;
       if (newValues == 0)
          throw new Fail();
       if (Values != newValues) {
          if (lastSaveFramePointer != currentFramePointer)
             Save();
          Values = newValues;
          foreach (var c in Constraints)
             c.Propagate(this);
       }
    }
```

```
        // Undo stack management
        private int lastSaveFramePointer = -1;
        private void Save() {
           variableStack.Push(this);
           valueStack.Push(this.Values);
           this.lastSaveFramePointer = currentFramePointer;
        }

        public static void RestoreValues(int FP) {
            while (variableStack.Count > FP)
              variableStack.Pop().Values = valueStack.Pop();
            currentFramePointer = FP;
        }

        public static int SaveValues() {
            return currentFramePointer = valueStack.Count;
        }

        // We use parallel stacks for variables and values to reduce GC load
        // You wouldn't do that for C++.
        private static int currentFramePointer;
        private static Stack<Variable> variableStack = new Stack<Variable>();
        private static Stack<UInt64> valueStack = new Stack<UInt64>();
    }
```

The Constraint class is nice and simple, in large part because it's abstract:

```
    public abstract class Constraint {
       public readonly List<Variable> Variables;
       public abstract void Propagate(Variable updatedVariable);
    }
```

Implementing constraints

Now we have everything we need to implement the actual propagate methods. In theory, there's a general way to do this. You iterate through all the variables. For each variable, you consider every possible value for the remaining variables that hasn't already been ruled out, and you determine which values of the first variable are consistent – given just this one constraint – with the remaining possible values of the other variables. Then you narrow to that set. If you didn't quite follow that, don't worry about it because it's not practical anyway; it would be a big nasty set of nested loops that would be too slow to run in-game. In practice, we use different *ad hoc* techniques for propagating different kinds of constraints, and we only support constraints for which there's a computationally reasonable propagation method.

Before we get into specific constraints, it's worth reminding you that we're representing our sets of possible values as bit masks. That means that we can do fast intersections, unions, and inverses with **&**, **|**, and **~**, respectively. It also means that we can use obscure bit-twiddling to optimize particular specialized set operations. An example of this is the code in IsUnique() above that tests whether a word has exactly one bit is set in a machine word.[2]

Equality constraints: $a = b$

As a pedagogical example, we'll start with equality constraints. You wouldn't necessarily do this in practice because the easiest way to enforce equality between two variables is to use one variable to represent both of them.[3] But if for some reason we didn't want to implement it that way, we could easily implement it using a propagate method. The basic idea is that if two variables have to be equal, then ruling out a value for one also rules it out for the other. So our propagate method is really simple:

```
public class EqualityConstraint {
   … constructor and stuff …

   public override void Propagate(Variable updatedVariable) {
      Variables[0].Narrow(Variables[1].Values);
      Variables[1].Narrow(Variables[0].Values);
   }
}
```

Remember that Narrow() doesn't set a variable's Values to its argument, it sets it to its argument AND the variable's current value. So a narrow call just means "rule out anything that isn't in the following set".

If you were really writing this code, you also wouldn't bother to update the variable that was just updated, so the code would really look like:

```
public class EqualityConstraint {
   … constructor and stuff …

   public override void Propagate(Variable updatedVariable) {
      if (updatedVariable == Variables[0])
         Variables[1].Narrow(updatedVariable.Values);
```

[2] If you really get into this stuff, it's worth reading Hacker's Delight, BitHacks, or some similar source for quick hacks for finding the number of 1 bits in a word, finding the most significant bit that's set in a word (aka the discrete logarithm), or the least significant bit set in a word (aka the find first set problem). If you really, really care deeply about performance, and you're writing in C, it's worth knowing that most modern architectures can compute these things in single instructions.

[3] If you're familiar with the unification algorithm, that's basically what the unification algorithm is about – dynamically merging variables together so that they get treated as one variable.

```
          else
             Variables[0].Narrow(updatedVariable.Values);
      }
   }
```

However, this makes the code harder to read, so we won't bother with that optimization for the rest of this section, except when it doesn't hurt readability.

Inequality constraints: $a \neq b$

Alternatively, we might want to enforce that two variables be different. In this case, the propagate method is pretty simple because it doesn't have anything to do until one of the variables gets narrowed to a unique value. The simple two-variable, no optimization version is just:

```
public class InequalityConstraint {
   ... constructor and stuff ...

   public override void Propagate(Variable updatedVariable) {
      if (Variables[0].IsUnique)
         Variables[1].Narrow(~Variables[0].Values);
      if (Variables[1].IsUnique)
         Variables[0].Narrow(~Variables[1].Values);
   }
}
```

Note the **~** inverts the bitmask, so that the Narrow() call is saying "this variable can have any value the (single) value of the other variable".

We can also extend this to a version that supports more than two variables and enforces that they are all different from one another:

```
public class InequalityConstraint {
   ... constructor and stuff ...

   public override void Propagate(Variable updatedVariable) {
      if (updatedVariable.IsUnique)
        foreach (var v in Variables)
           if (v != updatedVariable)
              v.Narrow(~updatedVariable.Values) ;

   }
}
```

Functional constraints: $a = \mathrm{f}(b)$

> **Important:** up until this point, we've ignored the issue of what the domains of our variables were, that is, whether they were numbers, days of the week, colors, etc.; we've assumed there was some way of mapping actual values to bit positions, and focused on manipulating bit sets with bitwise operations.
>
> From here on, we'll have to talk about specific values of variables. To simplify the presentation, we'll assume all variables are **small integers in the range 0…63**, and that these are **represented as bit sets in the obvious way**, i.e. the number i is in the set if bit number i is 1. To add i to a set, we OR it with 1<<i and to test whether i is in a set, we AND it with 1<<i and test whether the result is zero.

Another common constraint to require one variable is to be **some function of another variable** (or variables). If we think about the two-variable case, then the pseudocode for propagating $a = \mathrm{f}(b)$ is just:

```
$a$.Values = f($b$.Values)
$b$.Values = $\mathrm{f}^{-1}$($a$.Values)
```

where the f(Set) means the image of f under Set, i.e. the set { f(x) | x ∈ Set}, and f^{-1}(Set) is the inverse image, i.e. {x | f(x) ∈ Set}.

This corresponds to simple loops for update. That's less efficient than you'd like, but not necessarily the end of the world. Here's a version of the code for doing it. Again, for simplicity we assume that the variables of the variables are integers in the range 0...63. We also assume we have a pointer to the function f:

```
public class FunctionalConstraint {
   … constructor and stuff …

   Func<int,int> f;

   public override void Propagate(Variable updatedVariable) {
      var a = Variables[0];
      var b = Variables[1];

      if (updatedVariable == b) {   // compute a
         int image = 0;
         for (int i=0; i<64; i++)
            if (((1<<i)&b.Values) != 0) {   // i is in b.Values
               int result =f(i);
               image |= 1<<result;
            }
         a.NarrowTo(image);
      }
```

```
                else {    // updatedVariable == a. so compute b
                   int inverseImage = 0;
                   for (int i=0; i<64; i++) {
                     int result =f(i);
                     if ( (result & a.Values) != 0)    // result is in a.Values
                        inverseImage |= 1<<result;
                   }
                   b.NarrowTo(inverseImage);
                }
             }
          }
```

In practice, a lot of optimizations are possible here. Sometimes the image and inverse image can be computed through bit-shifting magic, table lookup, or just some clever piece of code. For example, if the function is $a = b + 1$, then this just amounts to narrowing a to b.Values<<1 or b to a.Values>>1. Even if you do have to write the loops, you can limit the iteration to values less than or equal to the largest current value of b.[4]

Functional constraints start to get expensive when they have more than one input, since it requires nested iterations. Update time is exponential in the number of arguments unless you have some clever way of reducing a given function to bit manipulation.

General relations: aRb

General relations are handled much like general functions: by looping over the possible values of one variable to find the possible values of another, then narrowing to the resulting set. This is easiest to do by precomputing lookup tables for all the values of one variable given the values of another; in the code below, these are called ATable and BTable, respectively:

```
public class RelationalConstraint {
   ... constructor and stuff ...

   UInt64[] ATable;
   UInt64[] BTable;

   public override void Propagate(Variable updatedVariable) {
      var a = Variables[0];
      var b = Variables[1];
```

[4] There are a couple of ways of doing this. One is to put b.Values in a temp variable and right shift the temp variable on each iteration of the loop, e.g. temp = temp>>1 (be sure to use a logical shift rather than an arithmetic shift). When temp is 0, you can stop the iteration early. The other approach is to compute the "discrete logarithm" of b.Values, which is just a fancy way of saying the number of the highest order bit that's set in the word. There are super clever ways of doing that. See Hacker's Delight or BitHacks.

```
            if (updatedVariable == b) {   // compute a
               int image = 0;
               for (int i=0; i<64; i++)
                 if (((1<<i)&b.Values) != 0) {   // i is in b.Values
                    image |= ATable[i];
                 }
               a.NarrowTo(image);
            }
            else {   // updatedVariable == a. so compute b
               int image = 0;
               for (int i=0; i<64; i++)
                 if (((1<<i)&a.Values) != 0) {   // i is in a.Values
                    image |= BTable[i];
                 }
               b.NarrowTo(inverseImage);
            }
         }
      }
```

Again, this can become ugly fast if you want to support unrestricted relations with more than two arguments. Then ATable, BTable, etc. become multidimensional arrays and the iterations become nested, and we need both exponential time and exponential space. So in practice, nobody does this.

Orderings: $a < b$

You often want to require that **one variable be less than another** in some ordering. This is easy to do provided the ordering of bits in the bit mask is the same as the ordering of values in the domain. Then we can do some magic bit hacking to construct a bitmask of all the bits to the left or right of another bit. Let MSB() and LSB() be procedures that take a bitmask and give you back a bitmask with just the most or least significant bit from the original set, respectively (most C compilers have intrinsics you can use for this). If you take an integer with only one bit set and you subtract 1, you get a mask of all the bits to the right of it, so we can write the propagate routine as:

```
      public class OrderConstraint {
         … constructor and stuff …

         public override void Propagate(Variable updatedVariable) {
            var a = Variables[0];
            var b = Variables[1];

            if (updatedVariable == b)
               // Rule out values for a >= the largest value for b
               a.NarrowTo(MSB(b.Values)-1);
            else {
               // rules out values for b <= the smallest value of a
               var lsb = LSB(a.Values);
               b.NarrowTo(~(lsb|(lsb-1)));
            }
         }
      }
```

Cardinality (count) constraints

Cardinality constraints are common in configuration or resource allocation tasks. They say that the **number of variables with some specified value** – call it the magic value – has to be in some specified range. So if we say the number of monsters in rooms is between 3 and 5, then we're saying that the number of variables with the magic value "monster" has to be at least 3 and no more than 5.

Our cardinality constraint class will look something like this:

```
public class CardinalityConstraint {
   ... constructor and stuff ...

   UInt64 MagicValueMask;  // Bitmask for the value we're constraining
   int LowerBound;         // Min number of vars allowed to have value
   int UpperBound;         // Max number of vars allowed to have value

   public override void Propagate(Variable updatedVariable) { ...  }
}
```

The propagate method needs to look at all the variables affected by the constraint and count:

- How many can **potentially** have the magic value, that is, how many haven't ruled it out yet, and
- How many **definitely** have the magic value, i.e. they've been narrowed to just the magic value.

The propagate method only responds when these counts hit their limits. If we're in danger of having too few (potential=LowerBound), we narrow every variable that can have the magic value to **only** the magic value. If we're in danger of having too many (definite=UpperBound), we exclude the magic value from every variable that isn't already committed to it:

```
public override void Propagate(Variable updatedVariable) {
   // Count possibles and definites
   int definite = potential = 0;
   foreach (var v in Variables)
     if ((v.Values & MagicValueMask) != 0) {
```

```
                possible++;
                if (v.Values == MagicValueMask)
                    definite++;
              }

          if (possible == LowerBound)
              // Make all possibles definite
              foreach (var v in Variables)
                  if ((v.Values & MagicValueMask) != 0)
                      v.Narrow(MagicValueMask);

          if (definite == UpperBound)
              // Exclude magic value for non-definites
              foreach (var v in Variables)
                  if ((v.Values & MagicValueMask) != 0 && v.Values != MagicValueMask)
                      v.Narrow(~MagicValueMask);
      }
```

A useful optimization for this case would be to have Narrow pass the previous Values of the variable along to Propagate. That would allow Propagate to quickly determine if the change to the variable required any action (i.e. if the variable had just become unique or had just ruled out the value entirely). Again, we haven't shown that version of the code so as to keep things simple.

The optimization you really want to do for this one is to cache the values of possible and definite, so you don't have to recompute them. Another is that as soon as either of the top-level if's in the propagate routine runs, the constraint is basically done – it can't ever apply more constraint to anything, and so it should stop running entirely. That said, implementing any kind of persistent internal state for the constraints is tricky because that state has to be saved and restored upon backtracking. While that's doable, it makes the code much more complicated.

Other optimizations and other implementation details

Now you have the basics of implementing a bare-bones constraint solver based on Macworth's AC-3 algorithm. In order to keep things simple, we've glossed over a lot of details, so now let's go back and talk about some of those details.

Randomization

If you're using constraint programming for games, there's a very high likelihood that you're using it because you want to general lots of different random solutions. However, the code we've given is completely deterministic. To randomize it, just modify the SolveOne() procedure to make random choices of variable and value[5]:

[5] To shuffle a set of numbers between 0 and k values, let p be the smallest prime $> k$ and let a be a random number in the range $0 \ldots p-1$ and s be a random number in the range $0 \ldots p-1$. Then just

```
**SolveOne**() {
  if all variables have exactly one value
    print the values and exit
  else {
    randomly pick a variable $v_i$ with more than one possible value
    for each $v \in$ Shuffle($v_i$.values) {      // Pick a value for $v_i$
      Narrow($v_i$, { $v$ })
      if no variable got narrowed to the empty set
        SolveOne()
      undo all the updates done in this iteration
    }
}
```

Constraint arcs and queued updates

A given variable might get updated many times in a given cycle of SolveOne(). So the real AC-3 algorithm keeps a queue of constraint/variable pairs that are waiting to be updated, and when when it wants to do an update, it first checks to see if there's already an identical update waiting in the queue. This can save a fair amount of work, but it requires complicating the backtracking mechanism: the queue needs to be saved and restored at choice points. Since in our work, the queue didn't significantly speed up our performance, we've left queuing out of the basic algorithm.

Variable ordering

We assumed that variables were chosen randomly for update, but there are a number of heuristics that can be used to speed up search. If we change SolveOne() so that it always chooses the variable with the fewest remaining possible Values, that will usually speed up solution time. If you break ties randomly between variables with equal numbers of remaining values, that will probably still give you a sufficient level of randomness to satisfy your player.

Representing domains

In the code above, we pretended all variables were over the domain of integers in the range 0…63. In real life, you'd probably represent your domains as enumerated types. You can then make the Variable class a generic class Variable<T> and instantiate it for different domains.

Advanced techniques

Now that you have a baseline finite-domain solver, here are some cool things you can add to support fancier tasks.

use the values a, $(a+s) \bmod p$, $(a+2s) \bmod p$, etc., skipping any values that are out of range (i.e. bigger than k). To shuffle the contents of an array, do the same thing, but take the ath element, the $(a+s) \bmod p$'th element, etc.

Interval methods

Finite domain solvers don't handle numbers well. You can handle integers provided they're small, but it's not as efficient as you'd like.[6] And floating-point values are basically impossible to handle.

For numeric domains, interval methods are an attractive alternative. The idea of interval methods is that instead representing the set of possible values for a numeric variable exactly, we approximate it as an interval, $[low, high]$. Then you can implement addition using the rule:

$$[l_1, h_1] + [l_2, h_2] = [l_1 + l_2, h_1 + h_2]$$
$$[l_1, h_1] - [l_2, h_2] = [l_1 - l_2, h_1 - h_2]$$

For more on interval arithmetic, see Wikipedia.

To support Intervals, you just make Variable an abstract class, with subclass Variable<T> that defines its Values property to be of type T. Now you can have Variable<UInt64> for finite domains and Variable<Interval> for floating-point quantities. Having done this, you can implement the functional constraint $a = b + c$ using the propagation method:

```
public override void Propagate(Variable updatedVariable) {
   var a = Variables[0];
   var b = Variables[1];
   var c = Variables[2];

   if (a != updatedVariable)
      a.Narrow(b.Values+c.Values);
   if (b != updatedVariable)
      b.Narrow(a.Values-c.Values);
   if (c != updatedVariable)
      c.Narrow(a.Values-b.Values);
}
```

where the arithmetic operators on Intervals are assumed to be overloaded to behave as specified above. You can easily generalize this to adding more than two variables, and it has the advantage that propagation time is linear in the number of variables added, rather than exponential as with the finite domain methods.

There are two downsides to interval methods, however. One is that multiplication and division are difficult to handle if variables are allowed to be zero or negative.

[6] For example, if you want to constrain one variable to be the sum of a bunch of others, then the update time is exponential in the number of things you're adding together.

The other is that SolveOne() can't exhaustively iterate through all possible values for a numeric variable because there are too many of them. In principle, you can handle this by doing binary search: you split the interval in half, try the lower half, and if that doesn't work try the upper half. You keep splitting recursively until you've split down to the resolution of your floating-point number system, at which point you've identified a unique float that is the value of the variable. People have done amazing things with this sort of technique.[7] However, in our work, we've tried to avoid directly solving for numeric variables, since it's quite expensive. In practice, when we've used numeric variables, they've been strict functions of finite-domain variables, so the system only needed to search the space of values for the finite-domain variables; the numeric variables, and their constraints then behave as filters that rejected unreasonable values for the finite-domain variables.

Path functions

Our original work with constraint propagation involved trying to do item and enemy placement for roguelikes.[8] This is basically a finite-domain problem; you assign a variable to each room to represent its contents, and add cardinality constraints to limit the numbers of enemies, health packs, etc.

Unfortunately, cardinality constraints don't put any limitations on how different kinds of items get placed relative to one another. So it's perfectly possible to get all the enemies placed at the beginning of a level, and all the resources you needed to defeat them (health, ammo, etc.) placed at the end.

To force the system to rationally interleave enemies and resources, you add extra variables to each room to represent the level of resource a typical player might have in that room. We find the value for those variables by reasoning about how resources would be acquired and used along different paths to that room, and so we call them path functions: the variables define for each room the minimum expected resource level over all "typical" paths a player might take to that room.

To know what typical paths would be, we assume we know for any room, which rooms the player might enter it from and which they would exit to when done in the room. That gives us a directed graph for the level where edge direction indicates the direction of forward progress through the level. This lets us compute how much ammo a player is "likely" to have as follows:

- For each room, we introduce three numeric constraint variables to represent (1) the amount of ammo the user is "likely" to have when they enter the room, (2) how much they're likely to acquire/use in the room, and (3) how much they're likely to have when they exit the room.

[7] See Van Hentenryck, Michael, and Deville, Numerica: A Modeling Language for Global Optimization.
[8] See Horswill and Foged, *Fast Procedural Level Population with Playability Constraints*. If you're interested, don't be intimidated by the gratuitous Greek letters. It's simple stuff and we're happy to explain it.

- The amount of ammo they're likely to have when they leave a room is the amount they're likely to have when they enter it, plus the amount they will likely acquire/use while in the room.
- The amount they're likely to have when they enter a room is the minimum of the amounts they'd have when exiting any of the rooms that come immediately before this room.
- Finally, the amount they'll likely acquire/use in the room depends on what's in the room, so it can be computed as a function of the contents of the room (which is already a constraint variable).

This gives us a system of constraint equations. Let i_r, o_r be the constraint variables for the ammo on entering and leaving room r, respectively, and Δ_r and c_r be the variables for the change in ammo and the contents of the room, respectively. Then we have that:

- $i_{\text{start}} =$ the player's initial ammo on entering the level
- $i_r = \min_{p \in \text{predecessors}(r)} o_p$ for any other room r

And for any room r:

- $\Delta_r = \text{f}(c_r)$ for some magic scoring function f, that the designer can tune to taste
- $o_r = i_r + \Delta_r$
- $o_r \geq 0$

Implementing this set of constraints allows you to detect and veto item placements that drive the player's ammo level unrealistically low in parts of the level, even if the total amount of ammo throughout the level is sufficient for the total enemies throughout the level. Tuning the thresholds and the scoring function allow you to adjust the difficulty of the game to taste, even if the levels are generated automatically.

Answer Set Programming

Answer Set Programming is a related declarative programming technique for finite-domain problems. ASP programs are written in a language with Prolog-like syntax but are solved by first automatically translating the program in a (very) large SAT problem[9], and then using a standard SAT solver to find the solution.

The disadvantage of ASP is that the translation into the SAT problem, known as "grounding," is quite expensive by game AI standards. When we implemented our path function solver in ASP, grounding times ran from 1.5 seconds to 34 seconds on a circa 2009 laptop and required up to 1.5GB of working set.

[9] A Boolean satisfiability problem. A SAT problem is a Boolean expression in terms of a set of (Boolean) variables one would like to make true. A solution to the SAT problem is a set of values for the variables that makes the overall expression true.

The advantage of ASP, however, is that a lot is known about how to solve very large, very hard SAT problems, and all this experience can be leveraged to solve other kinds of finite-domain constraint problems. ASP can solve **much** harder problems than the AC-3-based algorithm discussed here. For off-line problems (e.g. something in the build pipeline), ASP can be quite practical. ASP problems may even be practical to solve in-game in certain circumstances, provided they are grounded off-line as part of the build process and the SAT representation is loaded directly from the assets on disk. Be warned, though, it can still take tens of MB of RAM, so it may not be coming to consoles any time soon.

For examples, of cool things you can do for games using ASP, see the work of Adam Smith.

Conclusion: when should I use this?

Constraint programming is a great tool for solving certain kinds of combinatorial problems. It can be used for procedural content generation, configuration tasks, dynamic difficulty adjustment, level design tools, and many other kinds of problems that fit into the find-values-for-variables-subject-to-constraints model. Simple finite-domain solvers can have good performance properties, run in a small memory footprint, and can be written to work within existing game engines.

That said, the reader should be reminded that all constraint programming is a search problem and all search problems are **exponential in the general case**. Constraint propagation is intended to search the space more efficiently than the kinds of blind searches described at the beginning of this paper. For the right kind of problem, each successive choice made by the algorithm will constrain the next choice significantly, so that the algorithm never makes bad choices never needs to backtrack. In these cases, the algorithm runs in linear time (i.e. linear in the number of variables); but if constraint propagation doesn't rule out enough bad choices, then the algorithm devolves to blind search and we're back to exponential time, and the solver will not terminate before the patience of the user runs out, or in some cases, before the heat death of the universe.

The kinds of simple solvers discussed here are best used for **easy** constraint problems. Easy here means that there are **lots of solutions**, enough so that the algorithm is likely to stumble on one quickly, without having to backtrack much. If you're trying to solve a hard problem – one with a large search space and very few solutions, AC-3 won't work and you're better off using an industrial strength tool like a high end SAT solver.

Links

Our solver implementation (written in C#) can be found at https://code.google.com/p/constraint-thingy/