26

Rolling Your Own Finite-Domain Constraint Solver

Leif Foged and Ian Horswill

26.1 Introduction

Constraint programming is a kind of declarative programming. Rather than specifying *how* to solve the problem using some specific algorithm, the programmer provides a description of *what* a solution would look like. Then, a domain-independent search algorithm finds a solution using the description.

For example, suppose you are building a rogue-like or a dungeon crawler and you want to decide what items and enemies to put in what rooms. You probably know something about what you want in the rooms. You might want the number of enemies to be in a certain range and the amount of supplies to be in some other range, so that the level is balanced. You may also want to limit the number of goodies in a particular room, restrict certain key items to only appear in certain kinds of rooms like choke points, or outlaw having enemies in adjacent rooms.

You could probably write an *ad hoc* algorithm to do that and get it to work eventually, but it would take a fair amount of your time, time that could be better spent on other things. And it could have some very subtle bugs like making unsolvable levels once every 700 runs or going into an infinite recursion every 3000 runs. And you might get the algorithm debugged just in time for your design lead to tell you there's some new constraint you need to enforce, so you need to do it all over again.

Another example would be character or vehicle creation, either for generating NPCs or for use in the UI for the player character. You know a character needs some armor, a weapon, some skills, and so on, subject to some compatibility restrictions (e.g., evil mages can't wear the Divine Armor of Archangel Bruce) and some limit on build points. Again, you could roll your own algorithm, and it might be more efficient than an off-the-shelf algorithm for general constraint satisfaction, but it could be a time-consuming nightmare to debug. With the off-the-shelf algorithm, you already know it works. Provided it's fast enough for your needs, you can just code it up and move on to other issues.

This chapter will show you how to implement your own simple constraint solver. We will talk about the most common type of constraint satisfaction problem, finite-domain problems, and a simple and reasonably fast technique for solving them.

26.2 Simple Example

To simplify the presentation, we will use a simpler example than the aforementioned ones. Imagine you are generating a level for a tile game. There are 16 blocks and 6 possible colors for each block, as shown in Figure 26.1.

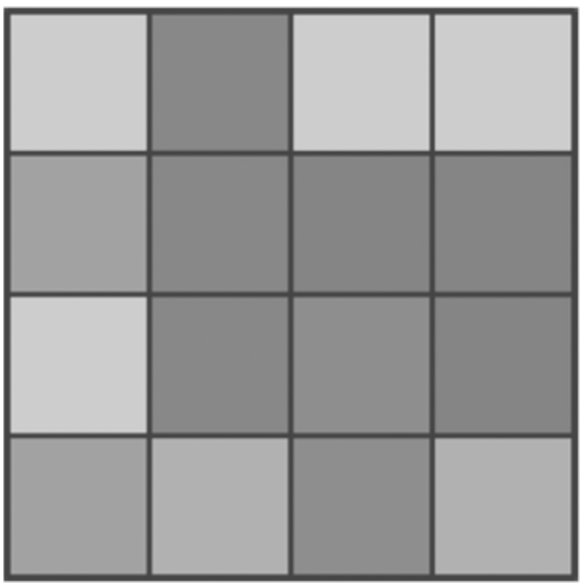

Figure 26.1

A possible level for our tile-based game.

And you need to select colors for the initial configuration, subject to some set of restrictions. For example, we might want to ensure that all six colors appear somewhere, no color appears on more than four tiles, and certain adjacent tiles have to have the same color. So we have a set of choices (the color of each individual block) and a set of restrictions on those choices. In constraint programming parlance, choices are known as *variables* and restrictions are known as *constraints*. The set of possible values for a given variable is called its *domain*. In this case, our variables have six different possible values (colors), so this is a *finite-domain* constraint satisfaction problem.

Finite-domain constraint satisfaction problems (CSPs) are the most widely studied and best understood constraint problems. In practice, the domains need to be not just finite, but small, so they tend to involve things like enums, booleans, or small integers. Finite-domain methods aren't practical for floats, even though there are technically only a finite number of 32-bit IEEE floats. Finite-domain CSPs are often used for solving configuration problems, such as generating puzzles or game levels. More broadly, they are useful for many of the things random number generators are used for in games today, but offer the ability to filter out classes of obviously bad outputs.

Here are some examples of constraints we can efficiently implement with a finite-domain constraint solver:

- Two specific blocks must have the same color.
- Three specific blocks must each have different colors.
- Some specific blocks must be blue.
- Some specific blocks cannot be green.
- Any two diagonal blocks must have different colors.
- At most four blocks of the same color.
- At least three red blocks.

All of these constraints share a common theme: as we *narrow* the possible values (colors) for one variable (block), we (often) also narrow the possible values of other variables involved in the same constraint. For instance, if two blocks are required to have the same color, then forcing one block to be red forces the other to be red as well. We will return to this idea shortly and discuss how exploiting the specific relationships between variables often enables us to dramatically speed up our algorithm.

We will structure the remainder of this article by starting with a simple brute-force algorithm and progressively making it faster and more sophisticated. Our primary focus is on making the code and explanations intuitive and clear, but we will occasionally shift gears to discuss practical performance optimizations.

26.3 Algorithm 1: Brute Force

There are 16 blocks and 6 possible colors for each block. For each block, we will assign a variable (`v1`, `v2`, etc.) to hold its color, as shown in Listing 26.1.

In the case of our block game, there are 16 possible tiles with 6 choices each, or 2,821,109,907,456 total possible levels. Even with relatively simple constraints, it may take quite a long time to discover an assignment of colors to blocks that satisfies them all.

Listing 26.1. An iteration of all possible assignments of colors to blocks.

```
foreach color for v1
   foreach color for v2
      foreach color for v3
          …
              foreach color for v16
                 if ConstraintsNotViolated()
                    return [v1, v2, v3, … v16]
```

Even worse, it's easy to find situations that will cause this algorithm to exhibit worst-case performance. Suppose we always try colors in the same order, say blue, green, yellow, etc. So the first color considered for any block is always blue. If we have the constraint that the first two blocks have different colors, that is, `v1 != v2`, then for the first time through the two earlier outer loops, we have `v1 = v2 = blue`, which is already a violation of the constraint. But the algorithm won't notice because it doesn't check for constraint violations until it gets all the way into the innermost loop.

Worse, when it does find the constraint violation, it won't jump back up to the outer loops, but rather blindly run all the inner loops to completion. So it will mechanically grind through all 78 billion possible color assignments for blocks `v3` through `v16` before considering a different color for `v2`, which was the actual source of the problem.

26.4 Algorithm 2: Backward Checking

There are some obvious optimizations to the brute-force approach. The simplest is **backward checking**. Each time we make a choice about a block's color, we verify that it's at least consistent with the other block colors we've chosen so far. If it's not, we can skip the evaluation of any additional nested loops and go directly to the next possible color for the current block. This gives us an algorithm like the one shown in Listing 26.2.

This approach is much better. It avoids the problem of exhaustively trying values of irrelevant variables when an already inconsistent set of choices has been made.

Listing 26.2. An iteration of all possible assignments of colors with backward checking.

```
foreach color for v1
   if ConstraintsNotViolated()
      foreach color for v2
         if ConstraintsNotViolated()
         foreach color for v3
            if ConstraintsNotViolated()
              …
                 foreach color v16
                    if ConstraintsNotViolated()
                       return [v1, v2, v3, … v16]
```

But unfortunately, it only checks that the choice we just made is consistent with the choices we've already made, not with future choices. Suppose instead of constraining the first two blocks to have different colors, we instead constrain the *first and last* blocks to be the same color and then add the constraint that no more than two blocks may be blue. Again, on our first pass through the two outer loops, the first two blocks get assigned blue. That doesn't violate any constraints *per se*. But since the first and last blocks have to have the same color, the last block must also be blue and will violate the two-blue-blocks constraint when we get around to assigning a color to the last block.

So choosing the first two blocks to be blue really does violate the constraints; the algorithm just won't detect it until it tries to assign a color to the last block. So once again, we end up trying 78 billion combinations before we give up and fix the bad choice we made at the beginning.

26.5 Algorithm 3: Forward Checking

To fix this new failure case, we need to detect when a choice we're making now precludes any possible choice for some other variable we have to assign in the future. More generally, we'd like to be able to reason about how the choices we've made so far *narrow* our remaining choices.

We can do this by explicitly representing the *set* of remaining candidates for a given variable rather than just a single value. We start by assuming any variable can have any value. When we choose a value for a variable, we're really just narrowing its set of possible values to a single value. If there is a constraint between that variable and some other variable, then we can usually narrow the set of possibilities for the other variable. This not only reduces the number of choices we need to explore but also lets us know when we've made an impossible set of choices.

Let's walk through an example in detail. To keep things simple, let's assume that we only have 4 blocks, so four variables, `v1`, `v2`, `v3`, and `v4`. Our constraints will be that

- Variables `v1`, `v2`, and `v3` must all have *different* values from one another
- Variables `v2` and `v4` must have the *same* value

We can represent this with the graph in Figure 26.2. Box nodes represent our variables, along with their possible values (R, G, B, C, M, and Y), circle nodes represent the constraints between them, and the edges specify which variables participate in which constraints.

As before, we start by picking a value for one of the variables. So, again, we assign `v1` the first color, blue (B), which means removing all other possible values, shown in Figure 26.3.

At this point, our previous algorithm would blindly pick a color for `v2`. In fact, it would make `v2` be blue, which violates the inequality (`!=`) constraint. But we can do better. We can *propagate* the value of `v1` through the graph to restrict the values of the other variables in advance. In this case, we know that `v1` can only be blue, but `v2` and `v3` have to be different from `v1`, so we can **rule out** blue for each of them, as shown in Figure 26.4.

But wait—there's more! Since `v2` and `v4` are joined by an equality constraint, `v4` can't have any value that `v2` doesn't have (or vice versa). So `v4` can't be blue either! This is shown in Figure 26.5.

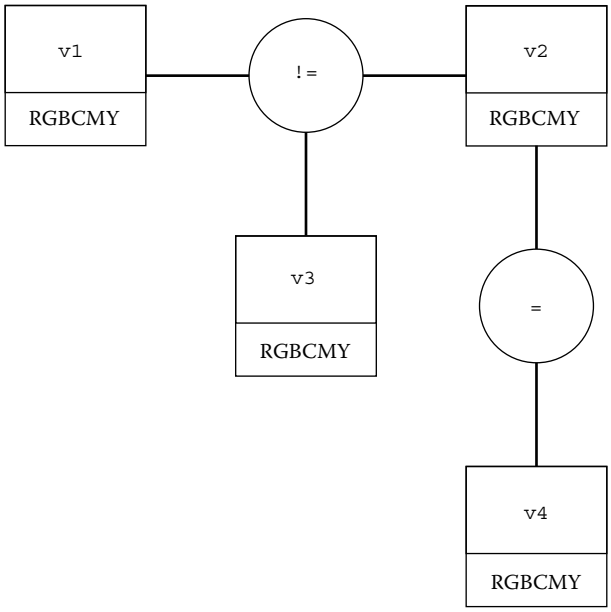

Figure 26.2

A graphical representation of the problem's variables and constraints.

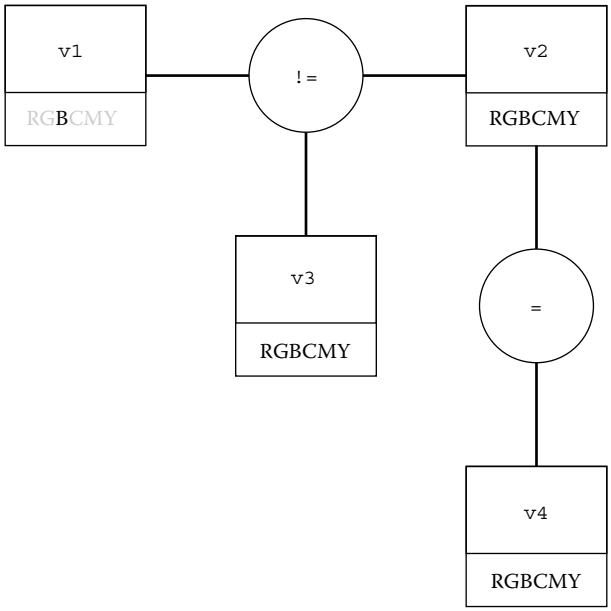

Figure 26.3

Variable `v1` is assigned blue (B), with all other color possibilities removed from `v1`.

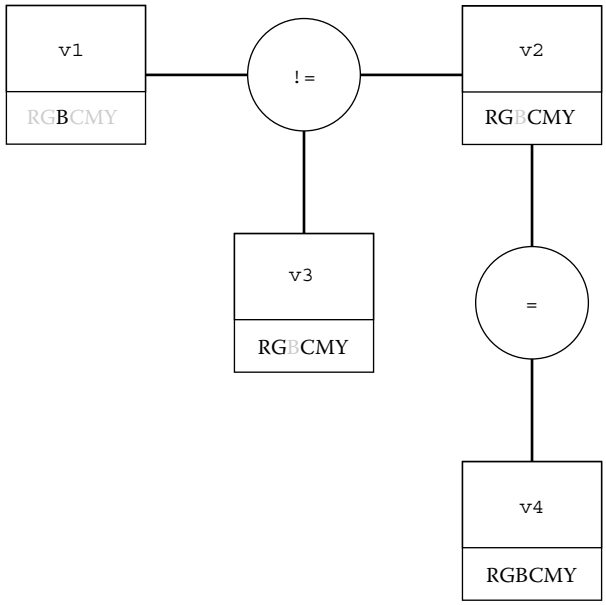

Figure 26.4

The value of `v1` is propagated through the graph, causing blue (B) to be removed from `v2` and `v3` due to the inequality constraint.

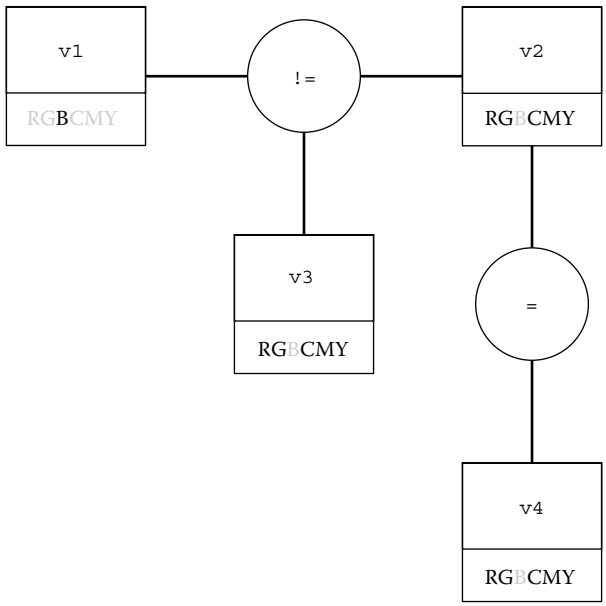

Figure 26.5

Constraints are further propagated to remove blue (B) from `v4`, due to the equality constraint between `v2` and `v4`.

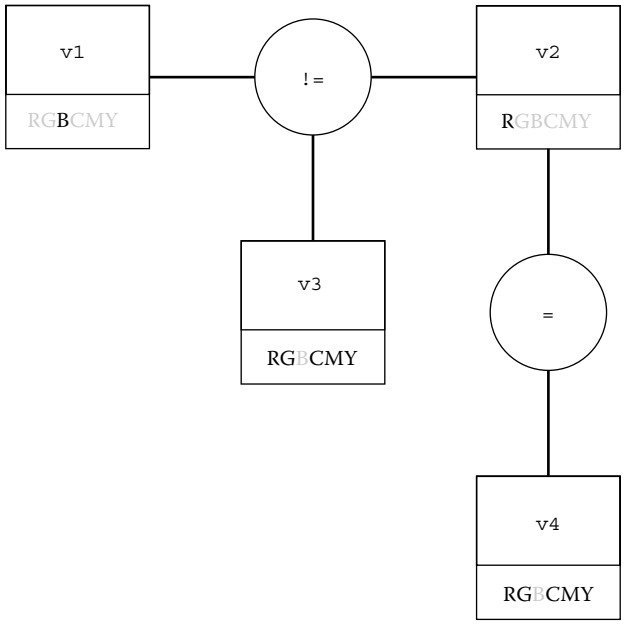

Figure 26.6

The variable `v2` is assigned red (R).

Now that there are no changes left to propagate through our constraints, we can resume our earlier algorithm and select a value for `v2`. Since we already removed blue (B) from its set of possible values, we will try red (R), shown in Figure 26.6.

Once again, we can propagate this choice through our constraints. Since `v4 = v2` and we just decided `v2` was red, we effectively forced `v4` to be red. Moreover, since `v3 != v2`, we can remove red from the possible values for `v3`, shown in Figure 26.7.

At this point, we just have one variable left, `v3`, and we've narrowed it to only four possibilities. We try one of them, say, green (G), shown in Figure 26.8.

Again, we propagate it through the constraint network. In this case, that means ruling out green as a possible color for `v1` and `v2`. But since we'd already ruled out green for each of them, we don't have to take any further action. So we know we have a valid solution.

26.6 Detecting Inconsistencies

That worked out very well for us, but here's an example where it doesn't work out as well. Suppose we're choosing colors for our four blocks, but we can only use three colors, and we have two different inequality constraints, such that every pair of blocks has to have different colors, except for `v1` and `v4`, shown in Figure 26.9.

This problem is perfectly solvable, but it requires choosing the same color for both `v1` and `v4`. Unfortunately, search algorithms don't know that. So suppose it chooses blue for `v1` and propagates (ruling out blue for `v2` and `v3`), shown in Figure 26.10.

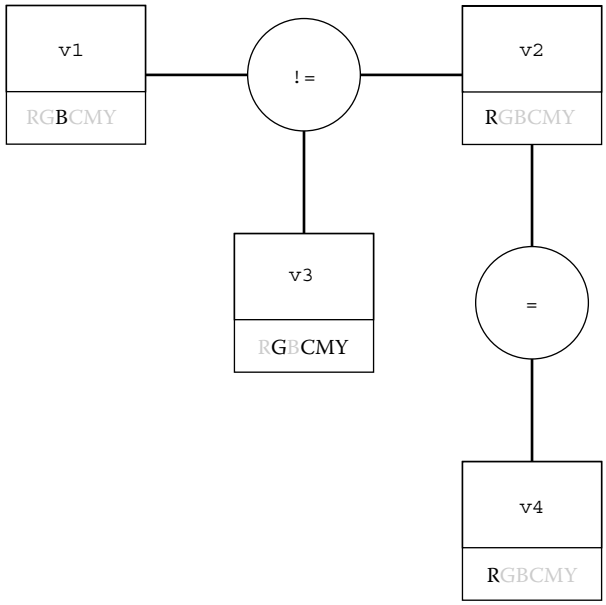

Figure 26.7

The variable `v4` is now red (R) due to the equality constraint between `v2` and `v4`. Further, red (R) can be removed from `v3`'s possibilities due to the constraint `v3` != `v2`.

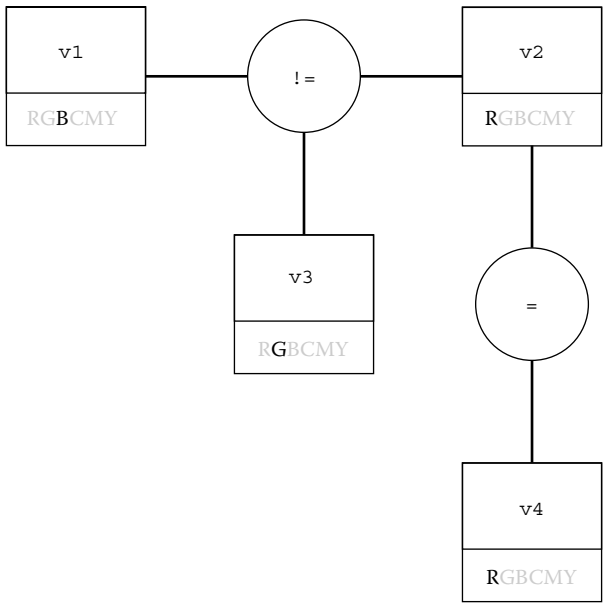

Figure 26.8

The variable `v3` is chosen as green (G). From previous steps, `v1` is blue (B), `v2` is red (R), and `c4` is red (R). This is a valid solution since it meets all constraints.

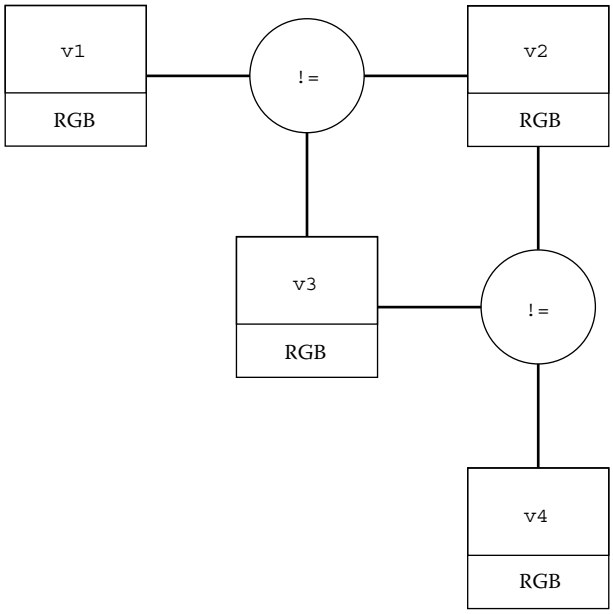

Figure 26.9

Only three colors allowed, with two different inequality constraints.

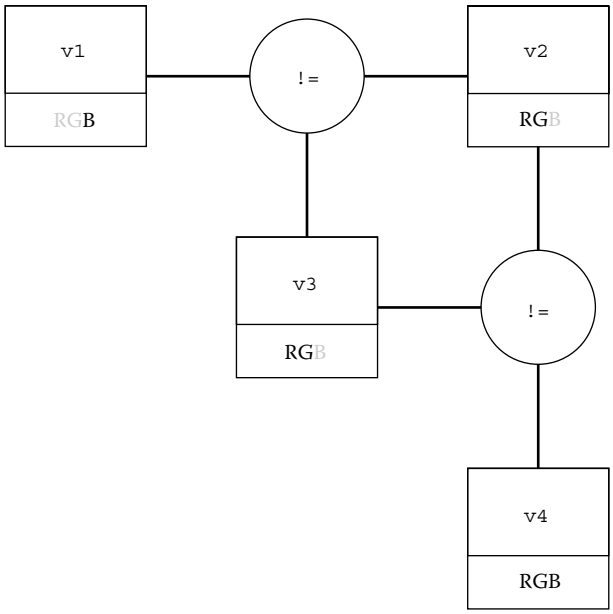

Figure 26.10

The variable `v1` is assigned blue (B) and the constraint is propagated such that blue (B) is ruled out for `v2` and `v3`.

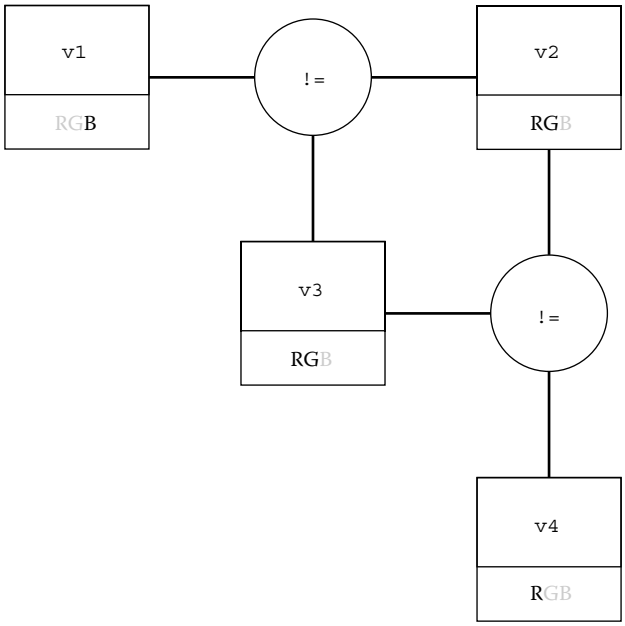

Figure 26.11

The algorithm choose red (R) for `v4`.

So far so good, but now suppose that for whatever reason, the algorithm chooses red for `v4`, as shown in Figure 26.11.

Now we have a problem when we propagate. First, `v4` tells `v2` and `v3` that they can't be red. But that means they've both been narrowed down to one choice (green), shown in Figure 26.12.

This violates both the inequality constraints, but the algorithm hasn't noticed this yet. Instead, it blindly continues to propagate the constraints. In this case, both `v2` and `v3` were just narrowed, so their narrowed values need to get propagated through the network. Let's say it propagates `v3` first. Since it's been narrowed to just green, it removes green from the possible values for the other nodes. That's not a problem for `v1` and `v4`, but for `v2`, green was all that was left, as seen in Figure 26.13.

We've narrowed the possible values for `v2` to the empty set, meaning there's no possible value for `v2` that's compatible with the values we've assigned the other variables. That means the last choice we made (assigning red to `v4`) *necessarily* leads to a constraint violation given the choices made before it (in this case, assigning blue to `v1`), and so we need to choose a different color for `v4`.

That's all well and good, except that in the meantime we've gone and changed the values of `v2` and `v3`. So when we decide `v4` can't be red, we need to set everything back to the way it was just before we chose red for `v4`. In other words, we need to implement *undo*.

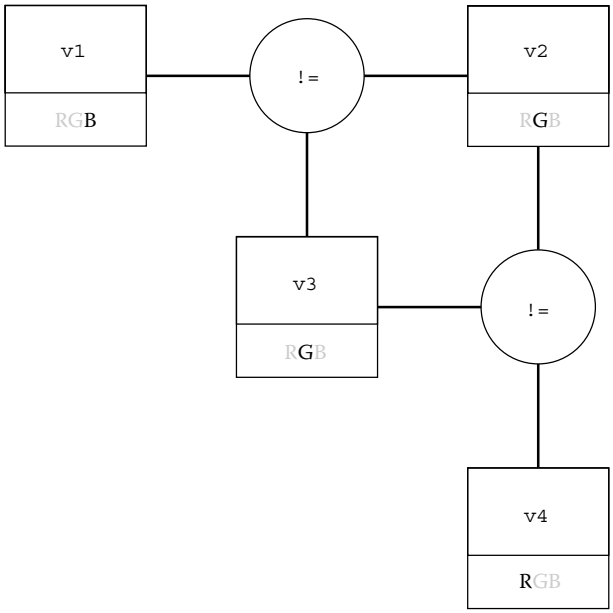

Figure 26.12

The variables `v2` and `v3` have been narrowed down to green (G), since they can't be red (R) due to `v4`.

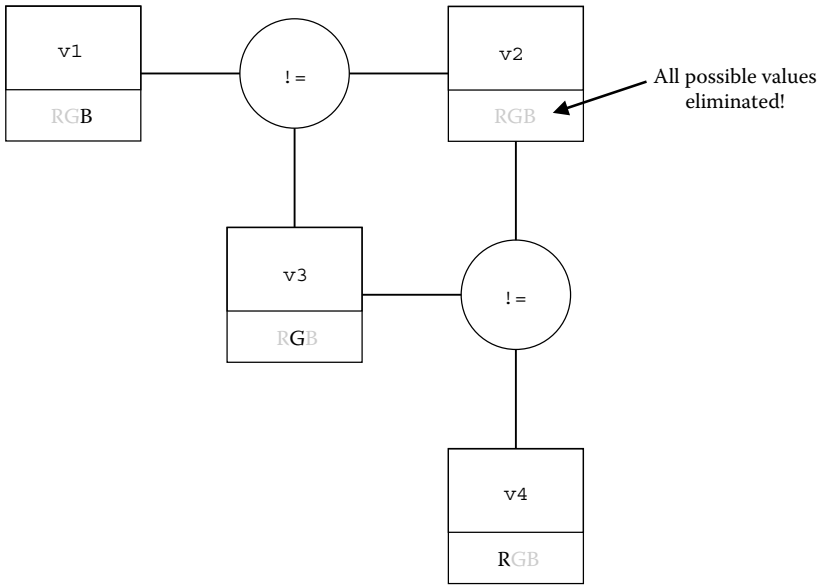

Figure 26.13

The variable `v3` was marked as green (G) and then propagated constraints, removing green (G) from `v2`. Now there are no possible values left for variable `v2`.

26.7 Algorithm 4: Forward Checking with Backtracking and Undo

Implementing undo for variables is pretty much like implementing undo for a word processor. We keep our undo information in a stack. Every time we modify a variable, we add an entry to the undo stack with the variable and its old value. When we want to undo a set of changes, we pop entries off the stack, setting the specified variables to the specified old values, until the stack is back to the depth it had had before we started changing things.

Listing 26.3 shows the basic outline of the algorithm for using constraint propagation: instead of just checking the constraints, we propagate updates through the constraints. If propagation ever narrows a variable to the empty set, it fails and returns false. We then undo any variable modifications we'd done and move on to try another value.

`PropagateConstraints` is a subroutine that takes a variable as input and propagates any changes through any constraints applied to that variable. Those constraints in turn further attempt to narrow other variables they are applied to and propagate those changes. So, as expected, this leads to a natural mutually recursive algorithm, shown in Listing 26.4.

This propagation algorithm is a variant of Mackworth's arc consistency algorithm #3 (AC-3) [Mackworth 77].

26.8 Gory Implementation Details

So, how can we take this conceptual idea of narrowing possible choices to variables and propagating constraints and implement it a programming language like Java or C#? A complete solver implementation is included on the book's website (http://www.gameaipro.com), and it's also hosted at https://github.com/leifaffles/constraintthingy. For the sake of brevity, we have tried to capture the main implementation strategies in the text, but the source code is much more liberal with comments and explanatory text.

An unfortunate side effect of our previous implementation is that creating new constraints involves modifying the solver's main propagation subroutine (in `Narrow`). Additionally, we'd like to make it possible to use the solver without having to remember to modify the undo stack and propagate constraints as we iterate through possible choices for blocks.

Listing 26.3. An iteration of possible colors with forward checking and backtracking.

```
foreach possible color for v1
   mark1 = undoStack.depth
   narrow v1 to just the selected color
   if PropagateConstraints(v1)
      foreach remaining color v2
         mark2 = undoStack.depth
         narrow v2 to just the selected color
         if PropagateConstraints(v2)
            foreach remaining color v3
                  …
         RollbackTo(mark2)
   RollbackTo(mark1)
```

Listing 26.4. The mutually recursive constraint propagation algorithm.

```
bool PropagateConstraints(variable v)
    foreach constraint c applied to v
        if !Narrow(c)
            return false
    return true

bool Narrow(constraint c)
    if c is an equality constraint:
        …
    else if c is an inequality constraint:
        …
    …

    foreach changed variable v:
        if !PropagateConstraints(v)
            return false
    return true
```

The most intuitive approach is to simply have a class for variables and a class for constraints. In the variable class, we will implement the main loop over the variables' possible values as an iterator over the finite domain in a method named `Colors`. Additionally, we will use a special method `SetValues` to narrow a variable's possible values, which will automatically save the old values on the undo stack and automatically propagate constraints. We demonstrate this in Listing 26.5.

Constraints will also be modeled as a class, as shown in Listing 26.6. They have an abstract method, `Narrow`, which each individual type of constraint will override to provide its own constraint-specific implementation of narrowing.

The simplest example of a constraint is equality. In practice, equality constraints are usually implemented by aliasing the two variables together, as is typically done in the unification algorithm. Then a change to one variable automatically changes the other. However, we write it here as a propagated constraint because it makes for a clear example.

As expected, implementing an equality constraint involves providing an implementation of `Narrow`. If there is a value that is in one variable's set of possible values and not in the other, it can *never* satisfy the constraint. So, any values that aren't in the *intersection* of the two variable's possible values shouldn't be considered. We implement this in `Narrow` in Listing 26.7 by computing the set intersection and calling `SetValues` on each value with this set (which, in turn, propagates these new values through more constraints).

A somewhat more complicated example is an inequality constraint. Like many constraints, inequality isn't able to narrow anything until one of the variables is narrowed to a unique value. But once one of the variables is constrained to a single value, we can definitively rule that value out for the other variable, shown in Listing 26.8.

One last example of a common constraint is a *cardinality constraint*. These constraints state that within some set of variables, at most (at least) a certain number of variables can (must) have a particular value. For example, in one of the preceding examples, we said that at most two blocks could be blue. We can implement such a constraint in a manner similar to the inequality constraint: we scan through the variables affected by the constraint,

Listing 26.5. Declaration of the variable class.

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

   public bool SetValues(FiniteDomain values) {
       if (values.Empty) {
         return false;
       }else if (Values != values) {
           UndoStack.SaveValues(this, _values);
           Values = values;
           if (!PropagateConstraints(this))
               return false;
       }
       return true;
   }

   IEnumerable<Color> Colors() {
       int mark = UndoStack.Depth;
       foreach (var color in Values) {
           if (SetValues(color)) {
               yield color;
           }
           UndoStack.RollbackTo(mark);
       }
   }
   …
}
```

Listing 26.6. Declaration of the constraint class.

```
abstract class Constraint {
   public abstract bool Narrow();
   …
}
```

Listing 26.7. Implementation of an equality constraint.

```
class EqualityConstraint {
   Variable a;
   Variable b;

   public override bool Narrow() {
       var intersection = Intersect(a.Values, b.Values);
       return a.SetValues(intersection) &&
              b.SetValues(intersection);
   }
}
```

Listing 26.8. Implementation of an inequality constraint.

```
class InequalityConstraint {
   Variable a;
   Variable b;

   public override bool Narrow() {
       if (a.IsUnique && !b.SetValues(SetSubtract(b.Values,
                                                  a.Values)))
             return false;
       if (b.IsUnique)
          return (a.SetValues(SetSubtract(a.Values,
                                          b.Values)));
   }
}
```

counting the number that has been constrained to just the specified value. If it's less than the maximum allowed, we don't need to do anything. If it's more than the maximum, we fail. But if it's exactly the maximum, then we remove the value from the candidates for any remaining variables, as shown in Listing 26.9.

An "at least" constraint is implemented similarly, but rather than monitoring how many variables can only have the specified value, it monitors how many haven't yet ruled it out. If that number goes below a threshold, then we fail, and if it reaches exactly the threshold, we force all remaining variables that can have the value to be exactly the value, shown in Listing 26.10.

Listing 26.9. Implementation of an "at most" constraint.

```
class AtMostConstraint {
   Variable[] variables;
   FiniteDomain constrainedValue;
   int limit;

   public override bool Narrow() {
     int valueCount;
     foreach (var v in variables)
        if (v.Value == constrainedValue)
           valueCount++;
     if (valueCount > limit)
        return false;
     else if (valueCount = limit)
        foreach (var v in variables)
           if (v.Value != constrainedValue
               && v.SetValue(SetSubtract(v.Values,
                                         constrainedValue))
               return false;
     return true;
   }
}
```

Listing 26.10. Implementation of an "at most" constraint.

```
class AtLeastConstraint {
   Variable[] variables;
   FiniteDomain constrainedValue;
   int limit;

   public override bool Narrow() {
     int valueCount;
     foreach (var v in variables)
        if (v.Value.Includes(constrainedValue))
           valueCount++;
     if (valueCount < limit)
        return false;
     else if (valueCount = limit)
        foreach (var v in variables)
           if (v.Value.Includes(constrainedValue)
               && v.SetValue(constrainedValue)
               return false;
     return true;
   }
}
```

26.9 Extensions and Optimizations

Surprisingly, this covers the basics of every part of the solver. You could load this code up and it would work. This section focuses on simple optimizations to the algorithm.

26.9.1 Finite-Domain Representation

We were pretty vague about how finite domains actually get implemented in the solver. These can be implemented any way you like subject to the restriction that operations on them have *value semantics*. While using a standard hash set data type such as C#'s `HashSet<T>` may seem appealing, it is ultimately impractical because such structures are expensive to copy (which must be done every time its value is modified since the undo stack must be able to restore a variable to a previous value at any point in time).

We strongly recommend implementing finite domains as fixed-size bit sets. For instance, for many problems, these can be implemented entirely with 32- or 64-bit integers (e.g., using a bit for each color.) This makes key operations like intersection extremely efficient to implement with bit-wise operations, as shown in Listing 26.11.

We have many of these operations implemented in the source code of our solver (in the `FiniteDomain` type). A great reference for bit hacks is the *Hacker's Delight* [Warren 12].

Listing 26.11. Implementation of set operations as bit-wise operations.

```
int Intersect(int a, int b) {return a & b;}
int SetSubtract(int a, int b) {return a & ~b;}
bool IsUnique(int a) {return a != 0 && (a & (a-1)) == 0;}
```

Listing 26.12. Implementation of queued narrowing operations ("constraint arcs").

```
class ConstraintArc {
   public Constraint Constraint;
   public Variable Variable;
   public bool Queued;
}

Queue<ConstraintArc> WorkList;

bool ProcessWorkList() {
     while(WorkList.Count > 0) {
           ConstraintArc arc = WorkList.Dequeue();
           arc.Queued = false;
           if(!arc.Constraint.Narrow(arc.Variable))
               return false;
     }
     return true;
}

class Variable {
    public bool SetValues(FiniteDomain values) {
       if (values.Empty) {
           return false;
       } else if (Values != values) {
           UndoStack.SaveValues(this, _values);
           Values = values;
           QueueConstraints();
       }
       return true;
   }

    IEnumerable<Color> Colors() {
       int mark = UndoStack.Depth;
       foreach (var color in Value) {
           if (SetValues(color) && ProcessWorkList()) {
              yield color;
           }
           UndoStack.RollbackTo(mark);
       }
   }
}

class EqualityConstraint {
   Variable a;
   Variable b;
   public EqualityConstraint(Variable a, Variable b) {
       this.a = a;
       this.b = b;
   }

   public override bool Narrow(Variable v) {
       FiniteDomain intersection =
           Intersect(a.Values, b.Values);
```

```
        if (v == a) {
            return a.SetValues(intersection);
        } else {
            return b.SetValues(intersection);
        }
    }
}
```

26.9.2 Constraint Arcs and the Work Queue

Another practical optimization is avoiding unnecessarily narrowing the same constraints and variables twice.

We create a global queue that we will use to queue up narrowing operations for constraints and variables. These queued operations are called *constraint arcs* because they represent the outgoing edges from constraints to the variables they touch. Representing these explicitly enables us to keep a bit on each constraint arc that we can test to see if the arc is already queued and avoid requeueing it. Listing 26.12 shows what this all looks like in code.

Further optimizations can be made by passing more information along in the queue. One possibility is to pass `Narrow` both the variable that changed and also its previous value. This can allow constraints like `AtMost` to determine when the change made to the variable is irrelevant to the constraint, since `AtMost` only cares when the specific value it's tracking is removed from the variable.

26.9.3 Randomized Solutions

Often games would like to be able to run the solver repeatedly and get different answers. So far, our algorithm has been completely deterministic. With a very small tweak and without compromising the ability of the solver to enumerate all solutions, we update our algorithm to iterate over the possible values of a variable in a random order:

```
foreach (var color in shuffle(Values)) {
    …
}
```

26.9.4 Variable Ordering

For more constrained problems, it's not uncommon to narrow a variable to the empty set (resulting in a failure) deep into the solver algorithm with a large undo stack. Unfortunately, if the responsible choice point was one of the first choices the algorithm made, it will take a long time before it gets around to reconsidering those initial choices because the initial assumption is that it was the *last* choice that was responsible.

This means that the *order* that we visit variables in can greatly affect performance. One option is to visit the most constrained variables before the least constrained variables. The intuition here is that more constrained variables are more likely to be narrowed to the empty set of values, and so we should figure that out up front instead of in a deep inner loop of the solver.

26.10 Conclusion

Constraint solvers are appealing for certain kinds of common tasks in procedural content generation (PCG) and game programming more generally. They let the programmer solve the problem with a minimum of fuss, and changes can be made to the constraints without having to modify and redebug some piece of custom code. That said, it must be stressed that constraint solvers ultimately rely on search algorithms and so can take exponential time in the worst case.

Simple constraint solvers such as the one described here are appropriate for use in game on "easy" problems, where the solver is being used to introduce variety by generating different random solutions each time the game is played. *Easy* problems have a lot of solutions, so the solver doesn't have to work very hard to find one. This usually means having relatively few constraints per variable. As the number of constraints per variable increases, the set of possible solutions usually decreases and the solver has to do a lot more work.

For "difficult" constraint satisfaction problems, that is, problems with large numbers of variables (large search spaces) but very few solutions, a more sophisticated solver, such as an answer-set solver [Smith 11], is more appropriate. However, these solvers, while capable of solving much more sophisticated problems, are generally designed for offline use. So they would be more appropriate for use in a design tool or in the build pipeline.
Happy hacking!

References

[Mackworth 77] Mackworth, A.K. 1977. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118.

[Smith 11] Smith, A.M. 2011. Answer-set programming for procedural content generation: A design-space approach. *IEEE Transactions of Computational Intelligence and AI in Computer Games*, 1:3.

[Warren 12] Warren, H. 2012. *Hacker's Delight.* Addison-Wesley Professional, Boston, MA.