Craft: a constraint-based random-number generator

Ian Horswill
EECS Department
Northwestern University
2133 Sheridan Road
Evanston, IL USA 60208
ian@northwestern.edu

ABSTRACT

In this paper I describe *Craft*, a floating-point constraint solver that generates sets of random numbers satisfying designer-specified algebraic constraints. Craft is available both as a C# API and as a Unity3D component that allows designers to directly specify constraints in the Unity editor. Despite the exponential complexity of the algorithm, the algorithm performs well on design-inspired problems at realistic scales.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications – Constraint and logic languages, nondeterministic languages, specialized application languages; D.3.3 [**Programming Languages**]: Language Constructs and Features – *constraints*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic – *Logic and constraint programming*; G.3 [**Probability and Statistics**]: Random number generation; I.2.1 [**Artificial Intelligence**]: Applications and expert systems – Games.

General Terms

Algorithms, Performance, Design.

Keywords

Procedural content generation, interval arithmetic, constraint satisfaction.

1. INTRODUCTION

Constraint solvers, and declarative methods more generally, are attractive for procedural content generation (PCG) applications. They allow a single system (the constraint solver) to be used for a range of problems, rather than handcrafting, testing, and debugging distinct algorithms for each separate class of problems. And designers can code simple cases even if they have relatively little programming knowledge.

Unfortunately, most existing constraint solvers are poor fits for game engines. The majority are finite-domain solvers, which have difficulty representing integer arithmetic, much less floating-point. Many high-end systems, such as SAT-solvers, trade high memory usage (on the order of hundreds of megabytes or more) for improved worst-case performance on very hard problems, and so are not currently feasible for use in-engine. Fortunately, many PCG problems are not hard instances in this technical sense, and so simpler, methods can suffice.

Proceedings of the 10th International Conference on the Foundations of Digital Games (FDG 2015), June 22-25, 2015, Pacific Grove, CA, USA.
ISBN 978-0-9913982-4-9.

In this paper, I describe Craft, a constraint solver that uses a combination of interval arithmetic and backtracking search with forward checking to efficiently solve a number of floating-point constraint satisfaction problems. From a user's standpoint, Craft looks like a random number generator that produces sets of random floats satisfying user-defined algebraic constraints.

Oddly enough, I can find no examples of similar systems in the literature. Most infinite domain constraint solvers focus on much more difficult problems, such as constrained optimization, where there is a very small set of solutions (often just one), and the goal is to enumerate all of them, or statistical sampling, such as rejection sampling [15], where the goal is to accurately reproduce a specified probability distribution. By contrast, in PCG applications, the admissible region is generally a large, connected region of the search space, and the solver is needed simply to generate variety (randomness). However, the exact probability distribution over the feasible region is often unimportant, so long as the sampling doesn't feel biased to the player.

1.1 Example

A simple build-point system for character creation can be specified in Craft as a set of variables to find values for (e.g. str, const, dex, int, wis, char, etc.), along with the desired range for each (e.g. 0-100, although 20-100 might be more sensible). The designer can then add to its list of constraints, such as a limit on the total number of build points:

$$\text{str} + \text{const} + \text{dex} + \text{int} + \text{wis} + \text{char} = 300$$

This says the designer doesn't care what the individual attributes are so long as they sum to 300 build points. If you want to make different attributes cost different numbers of build points, you simply multiply each by its cost, e.g.:

$$1.5\times\text{str} + \text{const} + \text{dex} + 1.2\times\text{int} + \text{wis} + \text{char} = 300$$

If the designer wants to enforce that the character be brawny rather than brainy, she can add the constraint:

$$\text{str} \geq \text{int} + 20$$

Which states that the strength has to be at least 20 points higher than the intelligence.

Craft can solve this problem in a little over a millisecond. However, this still leaves open the possibility for comically unbalanced characters where two attributes are at the maximum value and the rest are near the minimum. You can force a more balanced character either by adjusting the ranges for the attributes so as to forcibly rule out particularly low or high values, or by adding a limit on the variance of the attributes:

$$\text{var}(\text{str}, \text{const}, \text{dex}, \text{int}, \text{wis}, \text{char}) < 150$$

This enforces that the numbers cannot vary too much from their average value, at the cost of it taking much longer to solve (close to ¼ second). While one can certainly imagine handcrafting code to solve this problem manually, and that code might even solve the problem more efficiently, Craft allows one to solve problems

like this simply by entering the variables and constraints into the Unity editor.

2. Related work

There has been considerable interest in recent years in constraint programming methods for PCG. Gillian Smith et al.'s Tanagra system [20] used a finite-domain constraint solver to automatically generate partial levels for Mario-style platformers. Adam Smith and collaborators have used Answer Set Programming (ASP), a class of Boolean constraint solvers, in a variety of PCG problems [17–19]. And Horswill and Foged [8] used a finite-domain solver augmented with limited use of interval methods to perform automatic placement of items in dungeon levels for a roguelike.

All these systems performed their searches over finite-domain spaces, and indeed the bulk of the research on constraint programming has involved finite-domain problems. Constraint satisfaction on infinite domains (or finite approximations thereof, such as floating-point numbers) is considerably more difficult.

There has been some attempt to integrate constraint reasoning over the reals within the logic programming community. This has been motivated in part by the problematic treatment of arithmetic in logic programming languages such as Prolog, which essentially treat it as imperative rather than giving it logical (i.e. declarative) semantics. Various attempts have been proposed over the years to implement arithmetic in Prolog with proper logical semantics. CLP($\mathbb{R}$) [9] and ECLiPSe [2] delay execution of arithmetic constraints until sufficient variables have been instantiated to run a specialized numeric equation solver, such as a linear equation solver. Cleary [3] and Older and Vellino [13] augmented Prolog with a data type representing floating-point intervals; arithmetic expressions then create a constraint graph relating different interval quantities, allowing variables to be narrowed as other related variables are constrained. Although many of their use cases involved solving for ranges rather than distinct values, there were able to solve for distinct values of variables in polynomial equations using bisection search.

A considerable amount of attention in applied mathematics and numerical analysis has been devoted to the problem of constrained optimization. Here the problem is to find the global minimum for some cost function among the points satisfying a set of constraints known as the *feasible region* of the problem. In cases where the global minimum is non-unique, it is often desirable to enumerate all global minima. Linear optimization problems (aka linear programming) can be solved quite efficiently, but non-linear problems can be arbitrarily difficult to solve, depending on the nature of the constraints and cost function. One interesting and relevant case is the work of van Hentenryck et al. [5, 6], who used interval arithmetic combined with constraint propagation and branch-and-bound search to efficiently enumerate all local maxima/minima of a surprisingly wide range of constrained non-linear optimization problems. It is their work that largely inspired the work here.

Monte Carlo methods can also be used to pick random numbers based on constraints. In rejection sampling [11], one repeatedly picks random numbers until one is found within the feasible region. This has the advantage that one can sample not only from arbitrary feasible regions, but with arbitrary probability distributions over those regions. Unfortunately, the expected cost of rejection sampling is inversely proportional to the probability that a randomly chosen point in the sampled space will fall within the feasible region, and that probability can be arbitrarily low. For example, when sampling points on a (non-space-filling) curve embedded in a higher dimensional space, that probability is zero, meaning the expected execution time is infinite.

3. Overview of the algorithm

The algorithm used in Craft is conceptually quite simple. For each variable, we keep track of the interval (range) of possible values the variable can take. Our goal is to narrow those ranges until each variable has been narrowed to a single possible value. The core algorithm is:

```
While there are variables with non-unique values
  Pick a variable
  Narrow its range (somehow)
  Use the constraints to back-solve
    for the ranges of the other variables
  Backtrack if a variable's range is reduced to the empty set
```

When narrowing its range, we first try guessing a value in its current range, and if that fails, we try bisecting the range and continue. That's the basic algorithm. It's a non-deterministic algorithm, so there are a number of practical issues involved implementing non-deterministic algorithms (e.g. keeping an undo stack and rolling back changes during backtracking). For discussion of the issues involved in implementing non-deterministic algorithms see Norvig [12].

Sections 4 and 5 get into a number of fussy technical details in working out exactly how to back-solve for the possible values of variables, and in particular how to perform arithmetic over real intervals (or float intervals, as the case may be); some of those details can be dry to say the least. But the basic idea of the algorithm is actually quite simple. Suffice it to say that the reader who is not interested in reimplementing the algorithm may wish to skip to section 6, at least on the first reading.

4. Constraint satisfaction problems

A constraint satisfaction problem (V, D, C) is a collection of variables $v_i \in V$ that range over some set of domains $D_i \in D$, together with some set of constraints C that must hold between the variables. These constraints may be arbitrary relations.

4.1 Solving finite-domain CSPs

The simplest algorithm for solving CSPs is backtracking search, where different values are tried for the different variables in succession. A more efficient approach is backtracking search with a look-ahead strategy known as *forward checking*. Here, instead of assigning one value at a time to each variable, we instead track the set of possible values for the variable. This will initially be the variable's entire domain. When we try a given value for a given variable, we narrow its set of possible values to the singleton set of that one value, then back-solve for the possible values for other variables that are still compatible with that one remaining value for the variable we just narrowed. The search continues until all variables have been narrowed to singleton sets.

The advantage of tracking sets of possible values for variables rather than just a single current value is that it allows us to remove possible values from one variable when another is narrowed. Let R be some binary relation, with its left- and right-images defined as $R^l(S) = \{t \mid sRt, s \in S\}$ and $R^r(T) = \{s \mid sRt, t \in T\}$, respectively, and let V_i be the set of remaining possible values for variable v_1. Then if two variables v_1 and v_2 are restricted by a constraint $v_1 R v_2$ then each variable's values must be contained within the image of the other's variables, that is, $V_1 \subseteq R^l(V_2)$ and $V_2 \subseteq R^r(V_1)$. Since any narrowing of one variable potentially narrows its image under R, it also potentially narrows the other variable. And so any time one variable is narrowed, we can

updated the other using the rules $V_1 \leftarrow V_1 \cap R^l(V_2)$, when V_2 is narrowed, or $V_2 \leftarrow V_2 \cap R^r(V_1)$, when V_1 is narrowed. This may, in turn, allow further variables to be narrowed. The narrowing process continues until it reaches steady state. This is the basic idea behind Mackworth's AC-3 algorithm [10].

When solving CSPs using forward checking, one repeatedly selects variables to narrow, narrows them somehow (e.g. by restricting them to some singleton value), and propagating the narrowing to other variables. If at any time a variable is narrowed to the empty set, the algorithm backtracks. Otherwise, the search continues until all variables have been narrowed to unique values.

solve_fd(CSP)
  If all variables narrowed to singletons, then done
  Choose a variable v_i
  Choose a value $v \in D_i$ for v_i
    $V_i \leftarrow \{v\}$
    propagate(v_i)
    solve(CSP)

propagate(v_i)
  for each constraint of the form $v_i R v_j$
    Let $N = V_j \cap R^r(V_i)$
    if $N = \emptyset$ then fail
    if $N \neq V_j$
      $V_j \leftarrow N$
      propagate(v_j)
  for each constraint of the form $v_j R v_i$
    Let $N = V_j \cap R^l(V_i)$
    if $N = \emptyset$ then fail
    if $N \neq V_j$
      $V_j \leftarrow N$
      propagate(v_j)

This is the basic algorithm for arbitrary binary constraints with forward checking. The main points to understand are that the choice points occur at the choice of variable to narrow, and the value(s) to narrow it to, while failure is detected when the propagation of a constraint leads a variable to be narrowed to the empty set.

The case for constraints with higher-arity relations is defined analogously. However, in practice the propagate routine is often special-cased for different kinds of constraints.

Note that the propagate procedure implicitly assumes that a given variable appears only once in a given constraint. For constraints in which a variable appears multiple times, it can fail to prune variable values that are actually impossible. So the branching factor of the search will be higher than necessary in these cases.

4.2 Infinite domain CSPs

Solve_fd() implicitly assumes that the domains D_i of the variables are finite, both so that the sets V_i can be represented using a finite number of bits, and so that its iteration will run in finite time. So infinite domain CSPs require some modifications.

For infinite domains, and for practical purposes floating-point values might as well be infinite, the sets V_i can't be represented directly and so must be approximated. For numbers, sets of possible values S are typically approximated using the smallest closed interval $[a, b]$ that contains it. Similarly, we can't iterate over the elements of an infinite set in finite time. So instead of the for loop in the solve_fd() procedure, we use bisection search:

solve_infinite(CSP)
  If all variables narrowed to singletons, then done
  Choose a variable v_i
  Choose H to be one half of the interval V_i
    $V_i \leftarrow H$
    propagate(v_i)
    solve(CSP)

The exact way H is chosen is unimportant; the interval can be divided into halves and the haves tested in deterministic order, or a randomly chosen order, or even split at a random point.

4.2.1 Intervals over the extended reals

Although interval methods can and have been used for more exotic sets, we will use them here for intervals over the extended real numbers $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$. Using the extended reals has the advantage that we can treat infinite intervals such as the half-open interval $[0, \infty)$ as a closed interval $[0, \infty]$.[1]

We will define the smallest interval containing some set $S \subseteq \overline{\mathbb{R}}$ to be:

$$[S] = [\inf S, \sup S]$$

where inf and sup are the greatest lower bound, and least upper bounds of S, respectively. For most well behaved sets, these are simply the minimal and maximal elements of S. For infinite intervals, we will define them to be $\pm\infty$ where appropriate.

4.2.2 Closedness properties

Unfortunately, to make matters more confusing, we will be using two different senses of the word "closed" in this paper, that of an interval being closed, and that of a function being closed over the set of closed intervals over $\overline{\mathbb{R}}$. To say that a function f is closed over the (closed) intervals is to say that its image $\mathrm{f}([a, b])$ over a closed interval $[a, b]$ is itself some closed interval $[c, d]$:

$$\mathrm{f}([a, b]) \stackrel{\text{def}}{=} \{\mathrm{f}(x) | x \in [a, b]\} = [c, d]$$

Since the algorithms discussed here use only closed intervals, we will henceforth just say "interval" when we mean "closed interval" and use the term "closed" to mean the closure property of a function.

5. The algorithm

For true real numbers, solve_infinite() will never terminate, since bisecting true real intervals will never narrow them to singletons. For intervals over floating-point numbers, bisection will technically yield a singleton interval in a finite number of steps, albeit a painfully large number of steps. However, since our goal is not to enumerate all solutions, but rather to randomly choose a single solution, we can greatly speed the search by using a hybrid of the traditional forward-checking algorithm above with rejection sampling.

In rejection sampling, one first selects values for all the variables, then tests them against all the constraints. We can instead choose values for the variables from their current intervals one at a time, and then forward propagate them to narrow the intervals for other

[1] An interval is closed if it includes its endpoints. Interval arithmetic systems generally restrict themselves to closed intervals, since including open and half-open intervals would make for considerably more case analysis. However, for intervals over $\mathbb{R}$, an infinite interval has to be open because ∞ isn't an element of the reals. However, since it is an element of the extended reals, an infinite interval can be closed under the extended reals.

variables, thereby increasing the likelihood that we will select acceptable values for future variables:

solve_rejection(CSP)
  If all variables narrowed to singletons, then done
  Choose a variable v_i
  Choose $v \in V_i$
    $V_i \leftarrow \{v\}$
    propagate(v_i)
    solve(CSP)

While this algorithm is correct, and potentially faster than standard rejection sampling, it can still take arbitrarily long to find a solution. For example, given the CSP to find x, y such that $x^2 + y^2 = 1$, the algorithm might choose 0 for x, but then it can only narrow y to the interval $[-1,\ 1]$, which contains a very large number of floating-point values, of which only two are solutions to the constraint. Finding one of them can require an arbitrary number of guesses.

However, by adopting a hybrid strategy, where one first guesses, as in rejection sampling, then fails over to bisection search if the guess failed, we can get good performance on real problems. The resulting algorithm is:

solve_hybrid(CSP)
  If all variables narrowed to singletons, then done
  Choose a variable v_i
  N = choose(V_i)
    $V_i \leftarrow N$
    propagate(v_i)
    solve(CSP)

choose(V_i)
  First, choose $v \in V_i$ and try the set $\{v\}$
  If that fails, choose first one half of the interval V_i,
    and then the other half
  If both halves fail, choose() fails.

This is the algorithm used in Craft. If we follow its execution on the $x^2 + y^2 = 1$ example, the algorithm will first choose a value for one of the variables. Assume it chooses 0 for x. Then it narrows y to the interval $[-1, 1]$. It will first attempt to guess a value in that range, which will almost certainly fail. But having failed at its initial guess, it will then try to bisect the interval, yielding the interval $[0, 1]$. However, plugging this into the constraint $x^2 + y^2 = 1$, and given the knowledge that $x = 0$, the propagate() procedure can (see below) narrow y to the interval $[1, 1]$, i.e. deduce that $y = 1$.

5.1 Interval constraint propagation

Although constraints can in principle involve arbitrary set-theoretic relations, in practice they generally take the form of simple algebraic equations or inequalities over the variables; for example, stating that the average of some set of variables should lie within some range.

Constraints involving simple arithmetic operations are easily implemented, since the intervals are closed under addition, subtraction, and multiplication. The interval versions of the arithmetic functions are as follows:

$$[a,b] + [c,d] = [a + c, b + d]$$
$$[a,b] - [c,d] = [a - c, b - d]$$
$$[a,b] \times [c,d] = [\min S, \max S], \text{where } S = \{ac, ad, bc, bd\}$$

This allows us, given the intervals for the possible values of two variables, to easily determine the interval for the possible values of their sum.

5.1.1 Additive constraints

For addition constraints, i.e. constraints of the form $v_i = v_j + v_k$, we can propagate from one variable in the constraint to the others using the update rule:

$$V_i \leftarrow V_i \cap (V_j + V_k)$$
$$V_j \leftarrow V_j \cap (V_i - V_k)$$
$$V_k \leftarrow V_k \cap (V_i - V_j)$$

That is, for each for each of the variables, propagate() will compute the interval on the right, test whether it is:

- Empty, in which case, the algorithm backtracks
- Identical to the previous interval for the variable, in which case nothing happens, or
- Different from the previous interval for the variable, in which case the variable is updated, and propagate() is called recursively on the variable.

These operations in the update rules above are well defined, since $V_i, V_j,$ and V_k are all intervals and the intervals are closed under addition, subtraction, and intersection. The intersection operation on intervals is defined as:

$$[a,b] \cap [c,d] = \begin{cases} \emptyset, & \max(a,c) > \min(b,d) \\ [\max(a,c), \min(b,d)], & \text{otherwise} \end{cases}$$

5.1.2 Multiplicative constraints

For multiplication constraints, that is constraints of the form $v_i = v_j \times v_k$, we would ideally like to use the same approach:

$$V_i \leftarrow V_i \cap (V_j \times V_k)$$
$$V_j \leftarrow V_j \cap (V_i \div V_k)$$
$$V_k \leftarrow V_k \cap (V_i \div V_j)$$

However, intervals are not closed under division, since the quotient of two intervals can be the union of two intervals if the divisor interval contains 0. Due to space constraints, we give here a simplified version due to Ratz [14], as quoted in Hickey *et al*. [7] (see the latter for a more extensive discussion):

$$\frac{[a,b]}{[c,d]} = \begin{cases} [a,b] \times \left[\frac{1}{d}, \frac{1}{c}\right], & 0 \notin [c,d] \\ [-\infty, \infty], & 0 \in [a,b] \wedge 0 \in [c,d] \\ \left[\frac{b}{c}, \infty\right], & b < 0 \wedge c < d = 0 \\ \left[-\infty, \frac{b}{d}\right] \cup \left[\frac{b}{c}, \infty\right], & b < 0 \wedge c < 0 < d \\ \left[-\infty, \frac{b}{d}\right], & b < 0 \wedge 0 = c < d \\ \left[-\infty, \frac{a}{c}\right], & 0 < a \wedge c < d = 0 \\ \left[-\infty, \frac{a}{c}\right] \cup \left[\frac{a}{d}, \infty\right], & 0 < a \wedge c < 0 < d \\ \left[\frac{a}{d}, \infty\right], & 0 < a \wedge 0 = c < d \\ \emptyset & 0 \notin [a,b] \wedge c = d = 0 \end{cases}$$

Fortunately, it is sufficient for our purposes simply to form the least bounding interval of the underlying sets:

$$V_i \leftarrow V_i \cap (V_j \times V_k)$$
$$V_j \leftarrow \lceil V_j \cap (V_i \div V_k) \rceil$$

$$V_k \leftarrow \left[V_k \cap \left(V_i \div V_j\right)\right]$$

5.1.3 Odd integer power constraints

Odd integer power constraints, i.e. constraints of the form $v_i = \left(v_j\right)^n$ where $n \in \{1,3,5,\ldots\}$ are straightforward because intervals are closed over them:

$$[a,b]^n = [a^n, b^n]$$

Moreover, this mapping is invertible, so we can use the update rule:

$$V_i \leftarrow V_i \cap \left(V_j\right)^n$$
$$V_j \leftarrow V_j \cap (V_i)^{\frac{1}{n}}$$

Where inverse exponentiation for positive odd integers n is given by:

$$[a,b]^{\frac{1}{n}} = \left[a^{\frac{1}{n}}, b^{\frac{1}{n}}\right]$$

This is simple in the odd integer case because both the forward and inverse cases are strictly increasing functions.

5.1.4 Even integer power constraints

For even powers, the analysis is more complicated because even powers are not monotone, but the forward case, is still straightforward:

$$[a,b]^n = \begin{cases} [a^n, b^n], & a > 0 \\ [b^n, a^n], & b < 0 \\ [0, \max\{a^n, b^n\}], & \text{otherwise} \end{cases}$$

Inverting the mapping is problematic because, like division, $[a,b]^{\frac{1}{n}}$ is not necessarily an interval when n is even, but may be the union of two intervals. As with division, we use the least bounding interval for the inverse update:

$$V_i \leftarrow V_i \cap \left(V_j\right)^n$$
$$V_j \leftarrow \left[V_j \cap (V_i)^{\frac{1}{n}}\right]$$

Again, this involves a case analysis:

$$\left[V_j \cap (V_i)^{\frac{1}{n}}\right] = \begin{cases} V_j \cap (V_i)^{\frac{1}{n}}, & \min V_j \geq 0 \\ V_j \cap -(-V_i)^{\frac{1}{n}}, & \max V_j \leq 0 \\ V_j \cap \left[-(\max V_i)^{\frac{1}{n}}, (\max V_i)^{\frac{1}{n}}\right], & \text{otherwise} \end{cases}$$

5.1.5 Polynomial constraints

Given implementations of these basic arithmetic operations, we can implement arbitrary polynomial constraints by decomposing them and introducing new variables for the result of each operation. Thus $v_1 = v_2^2 + 1$ is decomposed into:

$$v_1 = v_3 + v_4$$
$$v_3 = v_2^2$$
$$v_4 = [1,1]$$

In performing this decomposition, it is useful to perform common subexpression elimination [1]. This is done by storing the results of previously computed functions in a hash table, indexed by the function computed and its arguments. When performing an operation, the hash table is first checked, and if an entry is found, the existing variable is used, otherwise, a new variable is created, constrained to be the result of the operation, and added to the hash table.

5.1.6 Monotone functional constraints

Constraints of the form

$$v_i = \mathrm{f}(v_j)$$

where f is any strictly increasing function, such as $\log x$, $\exp x$, $\tan x$, and positive square root, are easy to implement since, these are closed over intervals:

$$\mathrm{f}([a,b]) = [\mathrm{f}(a), \mathrm{f}(b)]$$
$$\mathrm{f}^{-1}([a,b]) = [\mathrm{f}^{-1}(a), \mathrm{f}^{-1}(b)]$$

Or for strictly decreasing functions:

$$\mathrm{f}([a,b]) = [\mathrm{f}(b), \mathrm{f}(a)]$$
$$\mathrm{f}^{-1}([a,b]) = [\mathrm{f}^{-1}(b), \mathrm{f}^{-1}(a)]$$

These constraints are implemented using the standard propagation rules:

$$V_i \leftarrow V_i \cap \mathrm{f}\left(V_j\right)$$
$$V_j \leftarrow V_j \cap \mathrm{f}^{-1}(V_i)$$

5.1.7 Piecewise monotone functions

All other continuous functions are at least piecewise monotone, meaning they can be decomposed monotonic segments. For example, the cosine function is monotone over all intervals $[\pi k, \pi(k+1)]$. So its image over an interval $[a,b]$ can be decomposed into:

$$\cos([a,b]) = \bigcup_k \cos([a,b] \cap [\pi k, \pi(k+1)])$$
$$= \bigcup_k \cos\left(\left[\max(a, \pi k), \min\left(b, \pi(k+1)\right)\right]\right)$$

where

$$\cos([\max(a,\pi k), \min(b,\pi(k+1))]) = \begin{cases} [\cos(\max(a,\pi k)),\ \cos(\min(b,\pi(k+1)))], & k \text{ odd} \\ [\cos(\min(b,\pi(k+1))),\ \cos(\max(a,\pi k))], & k \text{ even} \end{cases}$$

Although this is technically an infinite union, it will include only a finite number of nonempty terms whenever $[a,b]$ is finite. When $[a,b]$ is not finite, the union is $[-1,1]$ and so we don't need to compute it explicitly anyhow. Indeed, any time $b \geq a + \pi$ the result will be $[-1,1]$.

5.1.8 Statistical constraints

Simple statistical operations such as population mean and variance can be reduced to the constraints above. So:

$$\mathrm{mean}(v_1, \ldots, v_n) = (v_1 + \cdots + v_n)/n$$
$$\mathrm{stdev}(v_1, \ldots, v_n) = \sqrt{(v_1 - \mu)^2 + \cdots + (v_n - \mu)^2}$$

where $\mu = \mathrm{mean}(v_1, \ldots, v_n)$. Note that common subexpression elimination is important here.

5.1.9 Vector constraints

Similarly, constraints on vectors can be reduced to the scalar functions above. An n-vector is simply treated as n scalar variables and operations such as magnitude and dot product implemented in the standard manners:

$$\|\boldsymbol{v}\| = \sqrt{v_1^2 + \cdots v_n^2}$$
$$\boldsymbol{u} \cdot \boldsymbol{v} = u_1 v_1 + \cdots u_n v_n$$

where boldface variables are understood to denote vectors and subscribed italic variables their components.

5.1.10 Equality constraints

Although equality constraints of the form $v_i = v_j$ could in principle be implemented by adding a constraint to the constraint graph and explicitly propagating updates from one variable to another, it is considerably more efficient to treat them as the same variable. This is done using a preprocessing step when creating the original constraint graph. First we walk all the equality constraints to determine the equivalence classes of the variables using the union-find algorithm, and then we rewrite the remaining constraints to replace their variables with the representatives of their equivalence classes. This is essentially just a simplified version of Robinson's unification algorithm [16].

5.1.11 Inequality constraints

Inequality constraints are constraints of the form $v_i \leq v_j$.[2] Constraint propagation for these can be handled using the update rule:

$$V_i \leftarrow V_i \cap [-\infty, \min V_j]$$
$$V_j \leftarrow V_j \cap [\max V_i, \infty]$$

where max and min of an interval are understood to give the relevant endpoint of the interval.

5.2 Floating-point issues

One significant implementation issue is that to properly implement interval arithmetic, one needs to adjust the rounding mode of the floating-point unit, so that it rounds up when computing upper bounds for intervals, and rounds down, when computing lower bounds. This is properly implemented in standard interval arithmetic packages, such as the interval arithmetic in Boost C++.

Unfortunately, Craft was written to run inside Unity3D, whose virtual machine, the Common Language Runtime, does not support FPU rounding mode control. This means that it's possible for an interval arithmetic operation to produce an empty interval (one where the upper bound is lower than the lower bound) where it should have produced a finite or singleton interval, or for two interval arithmetic operations whose results should be identical to produce very close but non-intersecting intervals.

In the absence of FPU control, there was nothing to be done in Craft except to kluge around the issue by using double-precision computations, even though its API only returns single-precision values, and by using tolerance factors when testing for floating-point equality. So for example, the interval [1, 0.9999999999999] (an empty interval) is treated as the singleton interval [1,1].

Fortunately, this has not proven to be a problem for system performance.

6. Engine interface

To be most useful, a constraint solver should be able to run in-engine and be usable by designers without having to touch C# code. The current implementation includes an interface for Unity3D [21], a popular commercial game engine with a component-based architecture. The Unity interface presents Craft as a component that can be plugged into any game object. Variables and constraints can then be added to the component using the Unity editor (Figure 1). Designers can also specify fields of other components to which Craft's variables can be written back once Craft has found values for the variables.

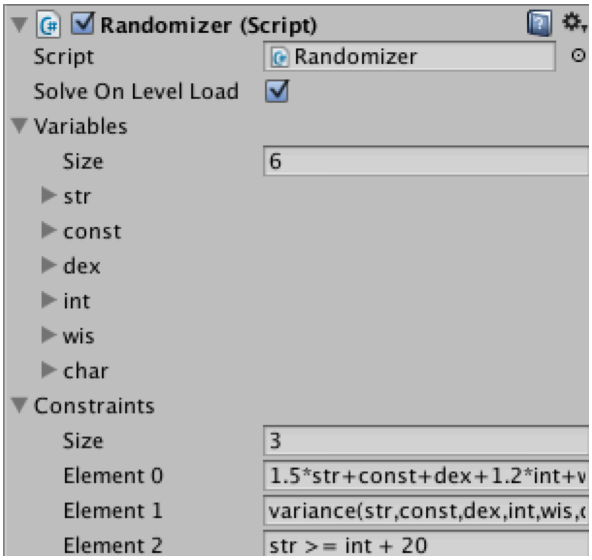

Figure 1: Editor interface for the Unity wrapper

Craft can be configured either to solve for the variables once at level load time, or can be called manually from script code to solve or re-solve.

The implementation of the Unity wrapper is straightforward. Constraints are entered by designers as strings in standard algebraic notation (e.g. "x^2+y^2<1" or "average(a,b,c)=7"), and then parsed using an operator-precedence parser [1] to build the constraint graph at level-load time. The constraint graph itself is not serialized, both because the string representation is more compact, and because of concerns about the efficiency of Unity's deserialization of graph structures.

The Unity interface also contains two other components. The first is a visualizer, that uses a CSP to drive a particle system, allowing the designer to visualize the sampling density of the CSP for up to three variables at a time. The second is a burn-in testing component that repeatedly solves a CSP and logs the solution and solution time to a spreadsheet.

7. Evaluation

Comparative evaluation of Craft is difficult because I'm not aware of any other systems that solve exactly the same problem. For example, there's little point in comparing its performance on the example in section 1.1, since the admissible region in that example is a measure zero subset of a six dimensional space and so rejection sampling's theoretical expected running time is infinite, even though in practice floating-point quantization would likely make it terminate in a finite, if impractically large, number of steps.

That said, Craft is very expensive relative to a normal pseudorandom number generator, and it generates results that are far less uniform than a traditional PRNG. So it is useful to at least try to characterize how much slower and how non-uniform so as to understand what kinds of applications it is suitable for.

7.1 Execution time

As with any search algorithm, Craft's worst-case execution time is exponential. For a search problem with n variables, c constraints, a floating-point representation with m bits of mantissa and e of exponent, the worst case running time of the algorithm is

[2] We cannot handle the strict inequality $v_i < v_j$ because it would require keeping track of both open and closed intervals; if $S < [0,1]$, then 0 cannot be an element of S and so $S \subseteq [-\infty, 0)$, which is not closed, even in the extended reals.

$O\left(3^{n(m+2^e)}c\right)$, which is comically far outside of the range of acceptable real-time performance. However, this is the performance in the worst-case scenario where the system essentially tests every possible set of floating-point values. The algorithm is based on the assumption that the CSPs it will be run on will have dense enough solution spaces that it will not have to search much. As mentioned above, rejection sampling has an infinite expected running time on the problem $x^2 + y^2 = 1$, but Craft will reliably find a solution in three steps of the search.

Table 1 gives the measured performance of the solver on a range of problems. The tests were performed on a 2012 MacBook Pro (Core i7 processor running at 2.3GHz, 8GB RAM) running a Microsoft Windows in a virtual machine (Parallels Desktop 10). Not surprisingly, simple problems run fast and complicated problems are much slower.

The results show a wide range of execution times depending on the complexity of the problem, from less than a microsecond, to nearly ¼ second. These are generally too slow for frequent use in the frame update loop of the game, and even then one might want to be given to a job system to run in a separate thread. But all are plausible for running during level-load, and trivial to do at design or build time.

Part of the explanation for the variation in run times lies in the fact that constraints represented as single algebraic equations can be unpacked into large numbers of internal constraints and variables. For example, the variance constraint $\mathrm{var}(a, b, c) = d$ unpacks into the constraints:

$$\frac{(a-\mu)^2 + (b-\mu)^2 + (c-\mu)^2}{4} = d$$
$$\frac{a+b+c}{4} = \mu$$

But these unpack into the primitive constraints:

$$a + b = t_1 \qquad b - \mu = t_6 \qquad t_7^2 = t_{10}$$
$$t_1 + c = t_2 \qquad c - \mu = t_7 \qquad t_8 + t_9 = t_{11}$$
$$\frac{t_2}{4} = \mu \qquad t_5^2 = t_8 \qquad t_{11} + t_{10} = t_{12}$$
$$a - \mu = t_5 \qquad t_6^2 = t_9 \qquad \frac{t_{12}}{4} = d$$

So that one variance constraint actually introduces 12 new variables and 12 primitive constraints.

In the case of the build point tasks, the "one-constraint" version actually involves 8 primitive constraints and 16 variables (of which 3 represent the constants 1.5, 1.2, and 300). The two-constraint version adds two extra constraints (total of 10) and three extra variables (total of 19). And for the version that includes variance, we add an extra 24 new constraints and variables so the apparently simple problem actually is decomposed into 34 constraints over 43 variables.

7.2 Uniformity

As both a speed and uniformity test, we used the particle system visualizer to drive to show the sampling density for the CSP $x^2 + y^2 \in [0.5, 1]$. The system generated and emitted 500 particles per second at 50FPS, producing a visualization of the uniformity of the sampling of the 2D space (see Figure 2). The frame rate appears to have been limited by the VM's video drivers rather than by CPU bandwidth.

Table 1: Average execution times

Task	**Avg μsec**
Scalar problems	
$a + b = 1$	<1
$q = a^2 + b$	5
$x^2 + y^2 \in [0.5, 1]$	18
Character build points $1.5{\times}s + k + d + 1.2{\times}i + w + c = 300$	1,152
$1.5{\times}s + k + d + 1.2{\times}i + w + c = 300$ $s \geq i + 20$	1,356
$1.5{\times}s + k + d + 1.2{\times}i + w + c = 300$ $s \geq i + 20$ $\mathrm{var}(s, k, d, i, w, c) < 150$	220,839
Vector problems	
Find perpendicular vectors $\boldsymbol{u} \cdot \boldsymbol{v} = 0$	9
Unit vector $\|\boldsymbol{u}\| = 1$	27
Orthonormal basis $\boldsymbol{u} \cdot \boldsymbol{v} = 0$ $\boldsymbol{v} \cdot \boldsymbol{w} = 0$ $\boldsymbol{u} \cdot \boldsymbol{w} = 0$ $\|\boldsymbol{u}\| = 1$ $\|\boldsymbol{v}\| = 1$ $\|\boldsymbol{w}\| = 1$	4800

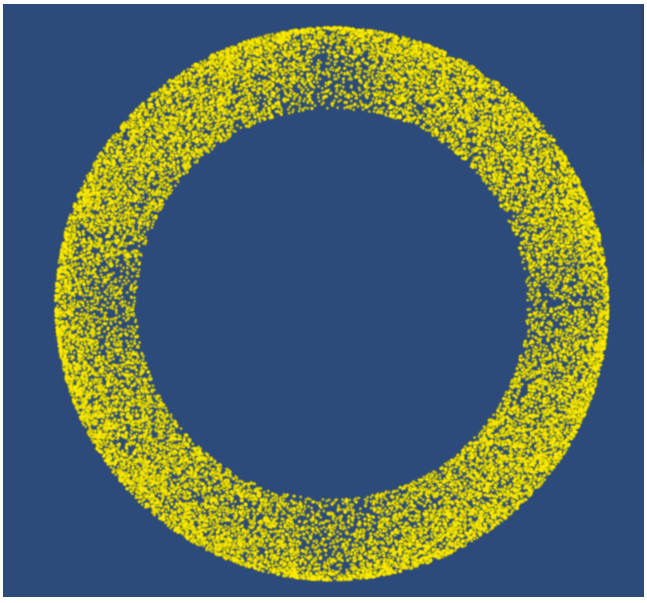

Figure 2: A particle system driven by Craft

As can be seen from the figure, density does indeed vary over the ring, with the highest densities being along the $x = \pm y$ lines and the lowest density being along the axes.

Although the density variation isn't dramatic, it is enough to be visible, which argues that Craft would not be suitable for problems such as placing large numbers of objects in a landscape, where the player would see very large numbers of instances at once. However, for applications where the player see only small numbers of instances at a time, the biases do not seem to be perceptible.

8. Future work

In principle, greater performance gains could be had by searching the space more intelligently. The current system uses a blind backtracking algorithm, and so some more intelligent algorithm such as backjumping and/or constraint learning could in principle greatly increase performance [4]. Whether this would be possible in practice remains to be seen.

A number of simpler optimizations could provide significant constant-factor improvements. For example, limitations in C# prevent efficient implementation of backtracking and control of floating-point rounding, so a C implementation could be noticeably more efficient. In addition, some constraints that are currently decomposed into smaller constraints, such as sums of more than two variables, could be implemented directly as primitive constraints, at the cost of potentially interfering with common subexpression elimination.

9. Conclusion

While constraint satisfaction is an attractive technology for PCG applications, most off the shelf constraint systems are offer poor performance on numeric problems, if they support them at all. Craft demonstrates that full floating-point constraint satisfaction can be solved efficiently enough to be useful for game applications. Although not appropriate to call on every frame, moderately complicated instances can be solved during level load, or jobbed out to a separate thread to handle relatively infrequent events such as the appearance of a new character.

In principle, custom code could be handcrafted to solve any specific problem for which one might use Craft. However, this would be an expensive (in programmer time) proposition, and one that would potentially have to be recapitulated each time a design change required a modification to the constraints being satisfied. Thus, even in cases where one might want to handcraft a solution for efficiency reasons, it may be preferable to use a general tool such as Craft during the initial exploratory stages of the design. The designer can then explore different possible constraints freely, and the programmer need only implement a custom solution once a given set of constraints and been committed to.

Acknowledgements

I'd like to thank Leif Foged, Robert Zubek, and the reviewers for their helpful comments and suggestions. Their attention is much appreciated.

REFERENCES

[1] Aho, A. V. and Ullman, J.D. 1977. Principles of Compiler Design (Addison-Wesley series in computer science and information processing). (Aug. 1977).

[2] Apt, K.R. and Wallace, M. 2007. Constraint Logic Programming using Eclipse. (Jan. 2007).

[3] Cleary, J.G. 1987. Logical Arithmetic. *Future Computing Systems*. 2, 2 (1987), 125–149.

[4] Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

[5] Van Hentenryck, P. 1997. Numerica: a modeling language for global optimization. *IJCAI'97 Proceedings of the Fifteenth international joint conference on Artifical intelligence* (Aug. 1997), 1642–1647.

[6] Van Hentenryck, P. et al. 1997. Solving Polynomial Systems Using a Branch and Prune Approach. *SIAM Journal on Numerical Analysis*. 34, 2 (Apr. 1997), 797–827.

[7] Hickey, T. et al. 2001. Interval arithmetic: From principles to implementation. *Journal of the ACM*.

[8] Horswill, I. and Foged, L. 2012. Fast Procedural Level Population with Playability Constraints. *8th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2012)* (Stanford, California, 2012), 20–25.

[9] Jaffar, J. et al. 1992. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*. 14, 3 (May 1992), 339–395.

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

[11] Von Neumann, J. 1951. Various techniques used in connection with random digits. Monte Carlo methods. *Nat. Bureau Standards*. 12, (1951), 36–38.

[12] Norvig, P. 1991. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann.

[13] Older, W. and Vellino, A. 1990. *Extending Prolog with Constraint Arithmetic*.

[14] Ratz, D. 1996. *On extended interval arithmetic and inclusion isotonicity.*

[15] Robert, C.P. and Casella, G. 2004. *Monte Carlo Statistical Methods*. Springer-Verlag.

[16] Robinson, J.A. 1971. Computational logic: The unification computation. *Machine Intelligence 6*. Edinburgh University Press. 63–72.

[17] Smith, A.M. et al. 2012. A Case Study of Expressively Constrainable Level Design Automation Tools for a Puzzle Game. *International Conference on the Foundations of Digital Games* (Raleigh, 2012).

[18] Smith, A.M. and Mateas, M. 2011. Answer Set Programming for Procedural Content Generation : A Design Space Approach. *IEEE Transactions on Computational Intelligence and AI in Games*. 3, 3 (2011), 187–200.

[19] Smith, A.M. and Mateas, M. 2010. Variations Forever : Flexibly Generating Rulesets from a Sculptable Design Space of Mini-Games. *IEEE Conference on Computational Intelligence and Games* (Copenhagen, 2010), 273–280.

[20] Smith, G. et al. 2011. Tanagra : Reactive Planning and Constraint Solving for Mixed-Initiative Level Design. *IEEE Transactions on Computational Intelligence, AI and Computer Games*. 3, 3 (2011), 201–215.

[21] Unity Technologies 2004. Unity 3D.