Cor3

Matthew Temple

Overview

Cor3 is a search/optimization algorithm I invented in 2009. It finds solutions to a multivariable problem without specific knowledge of the problem at hand. It's a genetic algorithm.

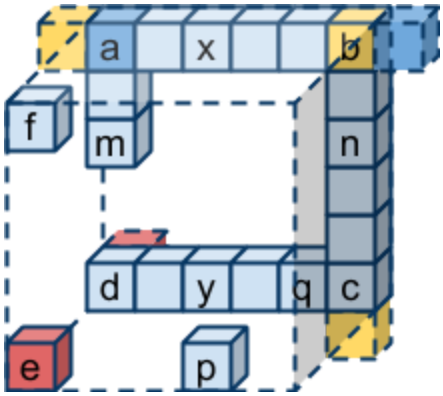
Reference implementation

http://github.com/clownfysh/cf/tree/master/inferno/cor3/

Structure

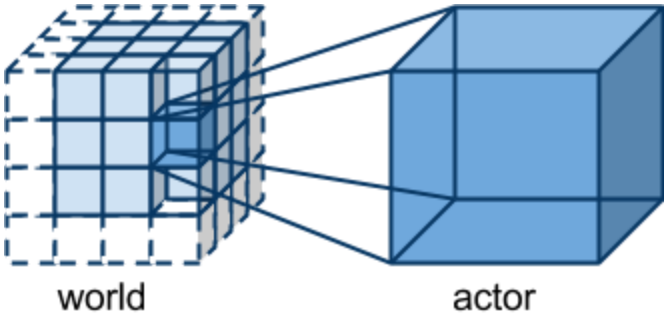
World

Cor3's world is a wrap-around cube whose logical elements are mapped onto a hypertorus-shaped surface, like this:

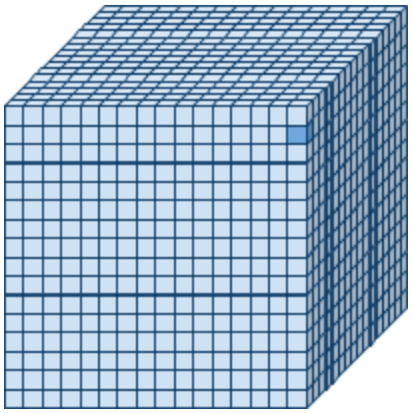


In this box, a is next to b, b is next to c, c is next to d, d is next to e, e is next to f and d, f is next to a, m is next to n, x is next to y, and p is next to q.

Each element of the cor3 world is called an actor.



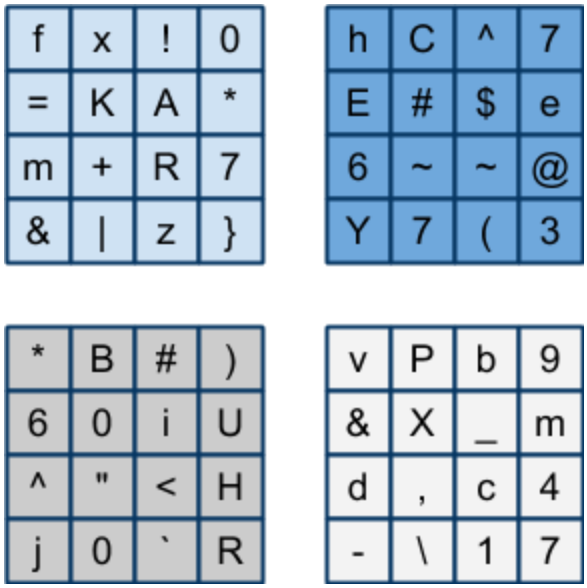
Each position in the world contains an actor. There are no empty positions. The world is $16 \times 16 \times 16$. There are 4096 actors in the world.



This world size is recommended. Changing it does alter some dynamics of cor3, but within certain ranges, changing the world size doesn't drastically help or hurt the algorithm.

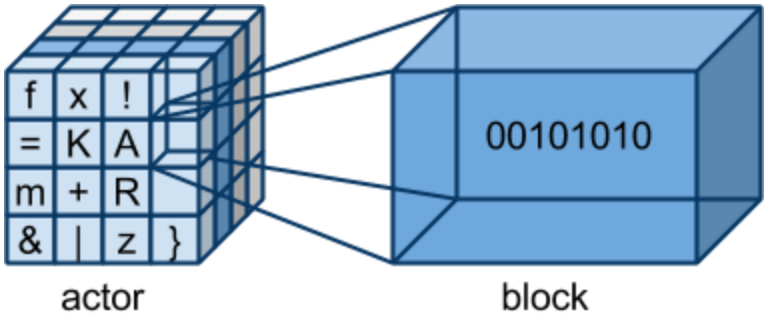
Actor

An actor contains 512 bits—or 64 bytes—of information.



This information is arranged in a $4 \times 4 \times 4$ block cube.

Each of those 64 blocks contains 8 bits—1 byte—of information.



This amount of information stored and this arrangement within an actor are recommended. The gist of the algorithm can be maintained while adjusting both of these, however.

Algorithm

Cor3 works with a scoring function. This function scores a solution to a problem, determining how good that solution is at solving the problem at hand. In my implementation, cor3's scoring function returns a real number—but that is incidental. Cor3 searches for solutions that either minimize or maximize this function.

Cor3 starts with a set of solutions you provide. If you don't provide initial solutions, it starts with random solutions.

Solutions are encoded as actors. Encode information about a solution to your problem in the bits of a cor3 actor. Encode them however you want. Use whatever precision and representation you like. Fields can overlap—bits can be overloaded.

The search is iterative. Each iteration is like this:

1. Select a random actor from the world.
2. Do the act function on that actor.

Decode information about solutions from the actors residing in the cor3 world and apply that information to your problem. As cor3 runs, its solutions improve.

Act function

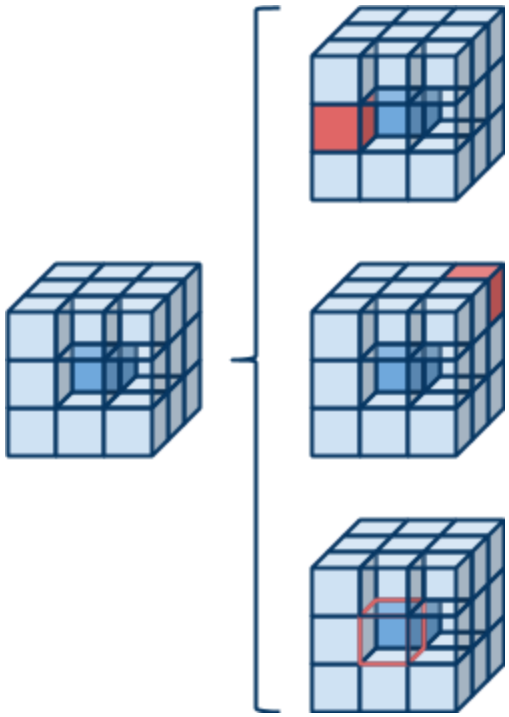
Anarchy modulus

The act function uses an anarchy modulus. The anarchy modulus a is a whole number greater than zero. $a = 4$ is recommended.

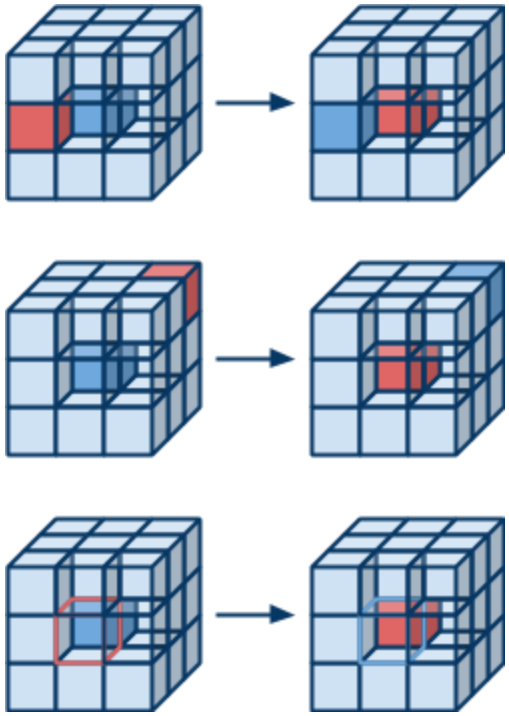
Algorithm

Act is a function called on a cor3 actor. For this *actor*:

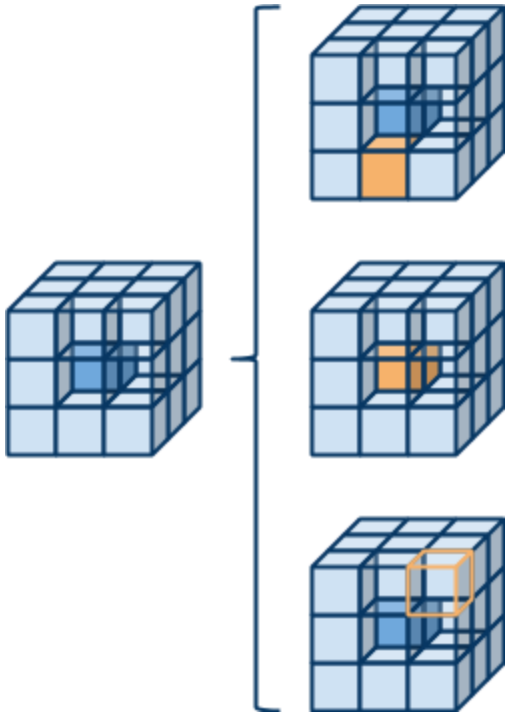
1. Find the **best nearby actor excluding this actor**. Consider the $3 \times 3 \times 3$ neighborhood of actors for which this *actor* is the center. Using the scoring function, find the actor in that neighborhood, excluding this *actor*, that has the best score (the highest or lowest score, depending on whether cor3 is maximizing or minimizing the scoring function). Call that *best_other_actor*. (In case of a tie, just pick one of the best.)



2. Swap, within the world, *actor* and *best_other_actor*.



3. Find the **worst nearby actor including this actor**.
   Consider the $3 \times 3 \times 3$ neighborhood of actors for which this *actor* is the center (this is a different neighborhood from the neighborhood considered in step 1 because this *actor* has moved, via step 2, within the world). Using the scoring function, find the actor in that neighborhood, including this *actor*, that has the worst score. Call that *worst_actor*. (In case of a tie, just pick one of the worst.)



4. Dance. In general, do $dance(worst\_actor,\ actor,\ best\_other\_actor)$. But, at a rate of $\frac{1}{a}$, do $dance(actor,\ best\_other\_actor,\ worst\_actor)$ instead.

Dance function

Mutation modulus

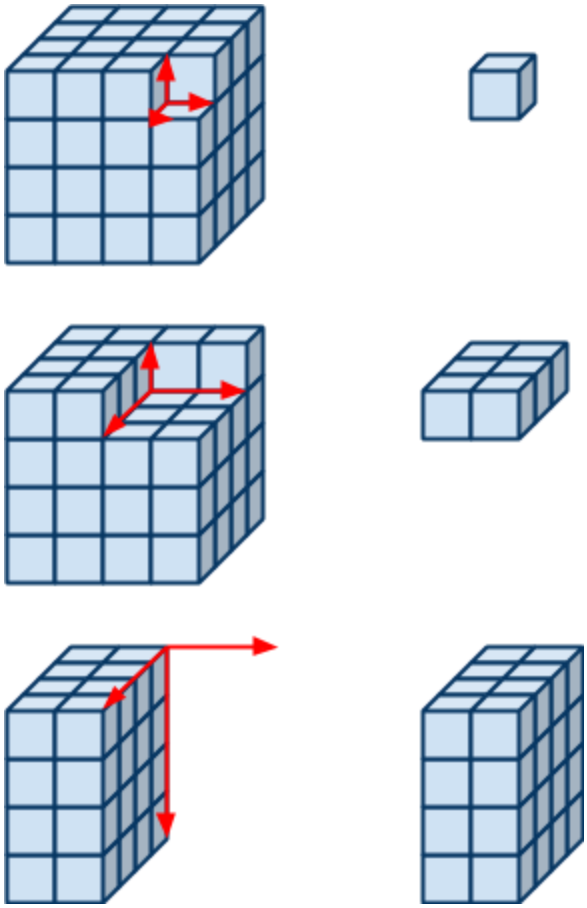
The dance function uses a mutation modulus. The mutation modulus m is a whole number greater than zero. $m = 64$ is recommended, based on the recommended actor size of $4 \times 4 \times 4$ blocks.

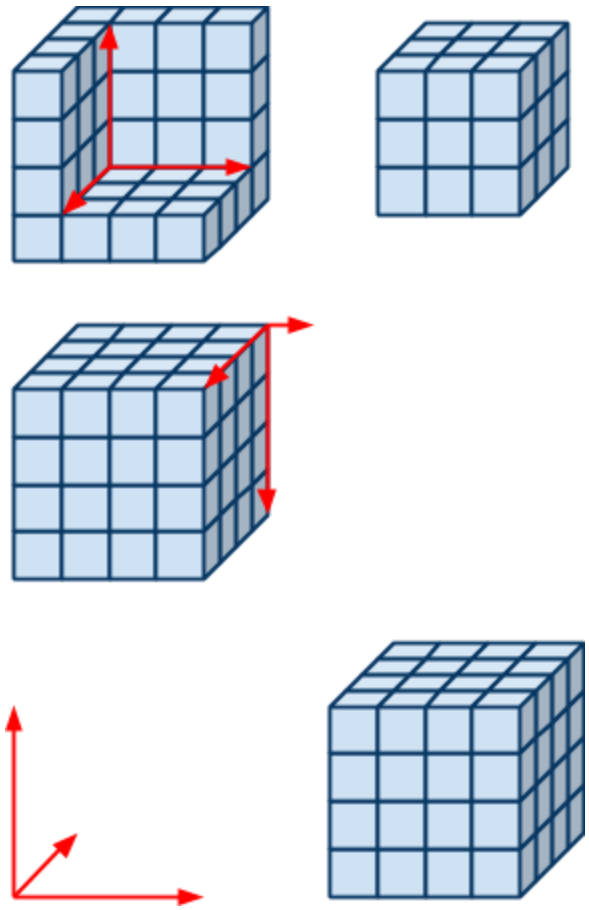
Algorithm

Dance is a function called on three actors. They're called $destination$, $source\_a$, and $source\_b$. To dance:

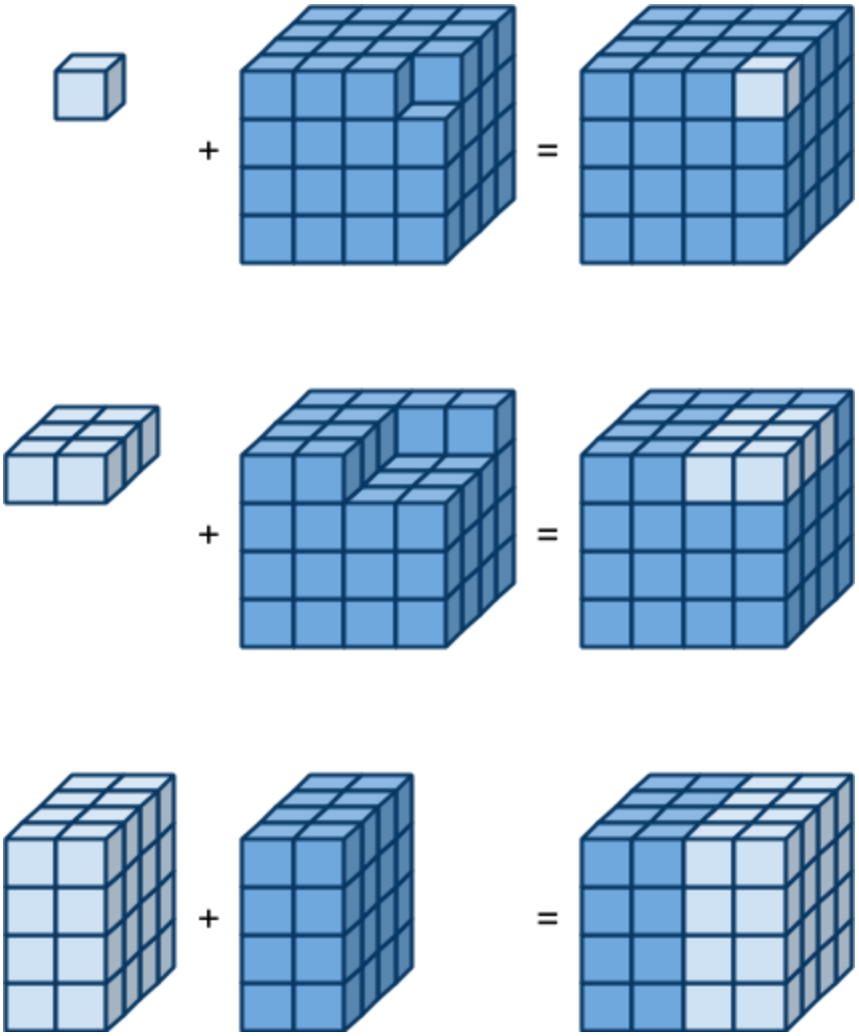
1. Choose a **cut point**. This is a 3-dimensional coordinate and 3 directions—a point lying on a vertex of blocks, and 3 discrete vectors within an actor. It slices the actor into two pieces. It slices off a block of the actor, possibly the whole actor, possibly nothing. It leaves a remainder, possibly the whole actor, possibly nothing. Choose the cut point

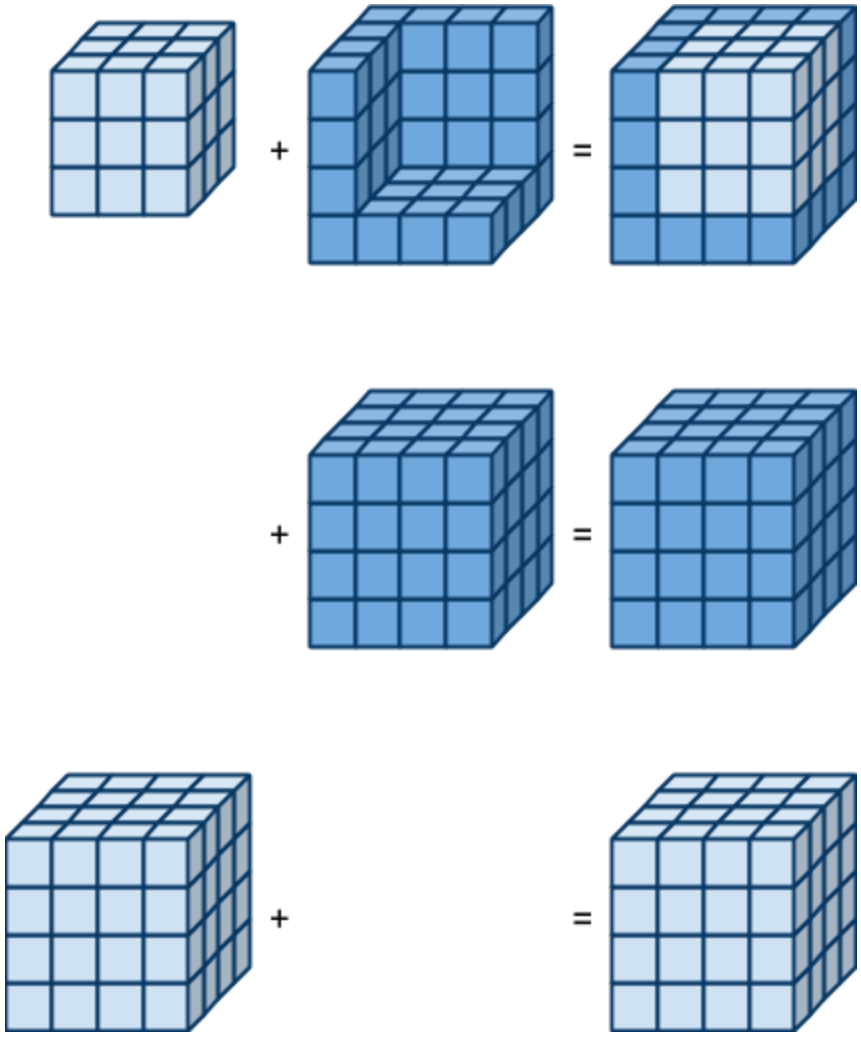
randomly.



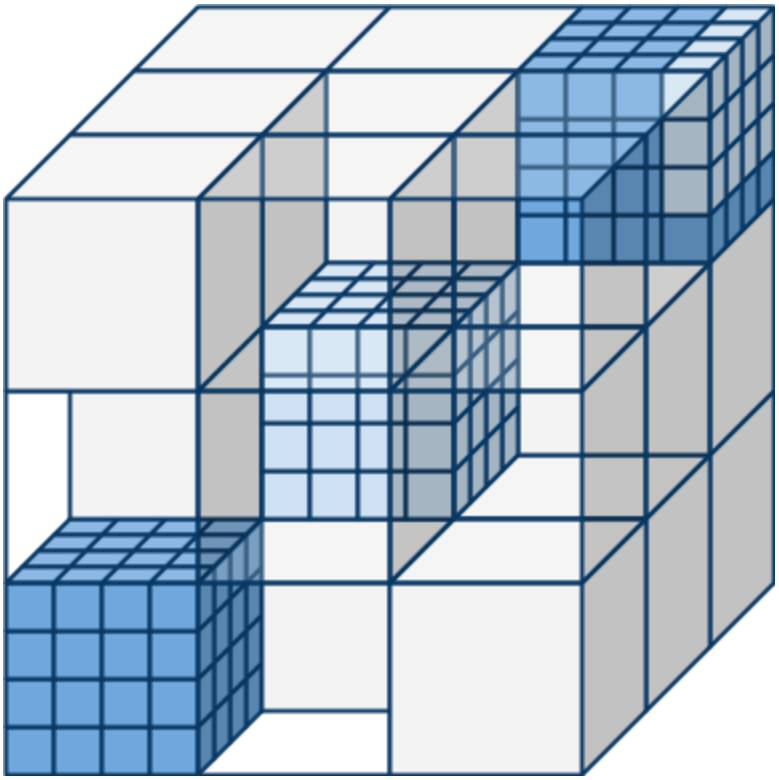


2. Perform 3d crossover.
    a. Slice off a block from $source\_a$ and $source\_b$ using the chosen cut point. Combine the sliced block of $source\_a$ with the remainder of $source\_b$.





b. Replace *destination* with the result of the crossover.



3. Perform **mutation**. This is done at the level of a block within an actor. For each block in $destination$, at a rate of $\frac{1}{m}$, replace the block's entire byte—all 8 bits—with a random byte.