Crossover in ROPER 2

May 4, 2018

Contents

1 Homologous Crossover in ROPER

The idea with the crossover mask (or, as its called in the source code, the `"xbits"`) mechanism is this: With each genome we associate a bitmask (for now, this is in the form of an unsigned, 64-bit integer, but this is a bit of a shortcut – we'd probably like to let it be as long as the longest possible genome). For the first generation, this value is initialized randomly. During crossover, the sites of genetic exchange are determined by *combining* (§1.1) the parents' crossover masks: crossover may occur (only) at those sites (mod 64) where the combined mask has a 1.

```
pub fn homologous_crossover<R>(mother: &Creature,
                               father: &Creature,
                               mut rng: &mut R) -> Vec<Creature>
where R: Rng, {
    let crossover_degree = *CROSSOVER_DEGREE;
    let bound = usize::min(mother.genome.alleles.len(), father.genome.alleles.len());
    let xbits = combine_xbits(mother.genome.xbits,
                              father.genome.xbits,
                              *CROSSOVER_MASK_COMBINER, rng);
    let child_xbits = combine_xbits(mother.genome.xbits,
                                    father.genome.xbits,
                                    *CROSSOVER_MASK_INHERITANCE, rng);
```

```
    let sites = xbits_sites(xbits,
                            bound,
                            crossover_degree,
                            &mut rng,
    );
    let mut offspring = Vec::new();
    let parents = vec![mother, father];
    let mut i = 0;
    /* Like any respectable couple, the mother and father take
     * turns inseminating one another...
     */
    while offspring.len() < 2 {
        let p0: &Creature = parents[i % 2];
        let p1: &Creature = parents[(i + 1) % 2];
        i += 1;
        let mut egg = p0.genome.alleles.clone();
        let mut sem = &p1.genome.alleles;
        for site in sites.iter() {
            egg[*site] = sem[*site];
        }
        let zygote = Chain {
            alleles: egg,
            metadata: Metadata::new(),
            xbits: child_xbits,
        };
        /* The index will be filled in later, prior to filling
         * the graves of the fallen
         */
        if zygote.entry() != None {
            offspring.push(Creature::new(zygote, 0));
        };
        if cfg!(debug_assertions) {
            println!("WITH XBITS {:064b}, SITES: {:?}, MATED\n{}\nAND\n{}\nPRODUCING\n{}\n**********
                     xbits,
                     &sites.iter().map(|x| x % bound).collect::<Vec<usize>>(),
                     p0, p1, &offspring[offspring.len()-1]);
        }
    }
    offspring
}
```

1.1 Combining the Crossover Masks

The precise means of combining the `xbit` vectors is left open to experimentation, but a good starting point seems to be bitwise conjunction (the `&` operator). This captures the intuition of restricting crossover to "genetically compatible" loci – loci at which the respective `xbits` of the parents coin-

cide. Of course, in the beginning, the high bits in the conjunction of the two masks means nothing at all, and has no real relation to genetic compatibility. It's just a scaffolding that, it seems, should have the capacity to *support* emergent compatibility patterns, or a sort of rudimentary speciation.

But for speciation to occur, the crossover masks should, themselves, be heritable. The masks of the two parents should be combined into a third, which will become the child's. This requires a second combination operator, with the same signature as the first, but we should guard against the temptation to use the same operator for both. It's pretty clear that `&` is poorly suited to play the role of an inheritance operator: within a few generations, the crossover masks would converge to 0s. We could experiment with other canonical boolean operators – `xor`, for example, or `nand` – that don't exhibit the fixed-point behaviour that `and` and `or` do, but the most natural choice might be to just use a secondary *crossover* operation to propagate the masks through the germ lines. One-point crossover seems like a poor fit, since it would disproportionately favour the first parent, but a simple, uniform crossover seems well suited to this task.

In addition, a slow and gentle mutation tendency should probably be incorporated as well: the crossover mask that the child will inherit, and share with its siblings, will be a uniform crossover of its parents', occasionally perturbed by a single bit-flip mutation.

```
fn xbits_sites<R: Rng>(
    xbits: u64,
    bound: usize,
    crossover_degree: f32,
    mut rng: &mut R,
) -> Vec<usize> {
    let mut potential_sites = (0..bound)
        .filter(|x| (1u64.rotate_left(*x as u32) & xbits != 0) == *CROSSOVER_XBIT)
        .collect::<Vec<usize>>();
    potential_sites.sort();
    potential_sites.dedup();
    let num = (potential_sites.len() as f32 * crossover_degree).ceil() as usize;
    if cfg!(debug_assertions) {
        println!("{:064b}: potential sites: {:?}", xbits, &potential_sites);
    }

    let mut actual_sites = rand::seq::sample_iter(&mut rng,
                                                  potential_sites.into_iter(), num).unwrap();
    if cfg!(debug_assertions) {
        println!("actual sites: {:?}", &actual_sites);
    }
    actual_sites
}
```

The utility functions referenced in the above block cover those methods of combination that can't be reduced to simple boolean arithmetic, but require the additional contribution of a pseudo-random number generator (PRNG). They are, in particular, fairly canonical bitwise crossover functions, as used in simple boolean genetic algorithms (GA). Their use introduces a secondary evolutionary process, supervening on the primary evolution of Return-Oriented Programming with ROPER (ROPER) creatures (return-oriented programming (ROP) chain payloads).

```
/// One-point crossover, between two u64s, as bitvectors.
fn onept_bits<R: Rng>(a: u64, b: u64, rng: &mut R) -> u64 {
    let i = rng.gen::<u64>() % 64;
    let mut mask = ((!0) >> i) << i;
    if rng.gen::<bool>() {
        mask ^= !0
    };
    (mask & a) | (!mask & b)
}

/// Uniform crossover between two u64s, as bitvectors.
fn uniform_bits<R: Rng>(a: u64, b: u64, rng: &mut R) -> u64 {
    let mask = rng.gen::<u64>();
    (mask & a) | (!mask & b)
}

fn combine_xbits<R: Rng>(m_bits: u64,
                         p_bits: u64,
                         combiner: MaskOp,
                         mut rng: &mut R) -> u64 {
    match combiner {
        MaskOp::Xor => m_bits ^ p_bits,
        MaskOp::Nand => !(m_bits & p_bits),
        MaskOp::OnePt => onept_bits(m_bits, p_bits, &mut rng),
        MaskOp::Uniform => uniform_bits(m_bits, p_bits, &mut rng),
        MaskOp::And => m_bits & p_bits,
        MaskOp::Or => m_bits | p_bits,
    }
}
```

2 Dependencies

For this to work, we'll need just a handful of dependencies: the pseudo-random number generator library in the `rand` crate, ROPER's own genotype structs in `gen::genotype` (along with the phenotype structures, for inessential reasons: it seems simpler for now to pass data in its phenome-wrapped

state, but this is a trivial implementation decision, and may change), and a few static parameter values that we essentially treat as immutable globals in this project, for the sake of convenience, sparing ourselves quite a bit of parameter clutter.

```
extern crate rand;
use self::rand::{Rng};
use gen::*;
use par::statics::*;
```

3 Putting it Together

```
<<crossover-module-dependencies>>

<<crossover-masks-utility-functions>>
<<combining crossover masks>>
<<homologous crossover>>
```