Contents

1 Homologous Crossover in ROPER

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

The idea with the xbits mechanism is this: Each genotype has an 'xbits' bitvector associated with it (in the form, for now, of a u64). For the first generation, this value is initialized randomly. During crossover, the sites of genetic exchange are determined by combining the two parents' xbit vectors: crossover may (or must?) occur (only) at those sites (mod 64) where the xbit vector has a 1. 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 coincide. 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.

This may lead to a few potentially interesting effects:

- facilitation of emergent homological crossover
- emergent speciation

We should have a float parameter $\text{crossover}_{\text{degree}}$, between 0.0 and 1.0, which select a certain ratio of the xover sites to use in a each particular crossover event.

```
/// 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,
    }
}
```

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

      let mut actual_sites = rand::seq::sample_iter(&mut rng,
                                                    potential_sites.into_it
      if cfg!(debug_assertions) {
          println!("actual sites: {:?}", &actual_sites);
      }
      (xbits, child_xbits, actual_sites)
  }
// test

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.allel
    let (xbits, child_xbits, sites) = xbits_sites(
        mother.genome.xbits,
        father.genome.xbits,
        bound,
        crossover_degree,
        &mut rng,
```

```
    );
    let mut offspring = Vec::new();
    let parents = vec![mother, father];
    let mut i = 0;
    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{}\n
                     xbits,
                     &sites.iter().map(|x| x % bound).collect::<Vec<usize>
                     p0, p1, &offspring[offspring.len()-1]);
        }
    }
    offspring
}
```